TURING 图灵程序设计丛书

/THEORY/IN/PRACTICE

软件之道

软件开发争议问题剖析

Making Software

[美] Andy Oram Greg Wilson 编著

鲍央舟 张玳 沈欢星 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS



图灵程序设计丛书

软件之道
——软件开发争议问题剖析

Making Software
What Really Works, and Why We Believe It

［美］Andy Oram　Greg Wilson　编著
鲍央舟　张　玳　沈欢星　译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

软件之道 : 软件开发争议问题剖析 / (美) 欧莱姆
(Oram,A.) , (美) 威尔逊 (Wilson,G.) 编著 ; 鲍央舟,
张玳, 沈欢星译. -- 北京 : 人民邮电出版社, 2012.3
(图灵程序设计丛书)
书名原文: Making Software
ISBN 978-7-115-27044-3

Ⅰ. ①软… Ⅱ. ①欧… ②威… ③鲍… ④张… ⑤沈
… Ⅲ. ①软件开发 Ⅳ. ①TP311.52

中国版本图书馆CIP数据核字(2011)第258450号

内 容 提 要

本书集合了几十位软件工程领域顶尖研究人员的实证研究，通过呈现他们长达几年甚至几十年的研究成果，揭示了软件开发社区普遍存在的一些确凿事实和虚构之事。书中探讨了更有效的编程语言，对比了软件开发人员之间的效率差异，验证了康威定理，并反思了软件行业的最新模式。本书将帮助读者拓宽视野，更好地选择适合的工具和技术，并最终成为一名更加优秀的软件行业从业人员。

本书适合所有软件开发人员和研究人员阅读。

图灵程序设计丛书

软件之道：软件开发争议问题剖析

◆ 编著　[美] Andy Oram，Greg Wilson
译　鲍央舟　张　玳　沈欢星
责任编辑　傅志红
执行编辑　李　盼

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：28.5
字数：728千字　2012年 3 月第 1 版
印数：1－3 500册　2012年 3 月北京第 1 次印刷

著作权合同登记号　图字：01-2010-8061号

ISBN 978-7-115-27044-3

定价：89.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

O’Reilly Media, Inc.介绍

O’Reilly Media通过图书、在线服务、杂志、调查研究和会议等方式传播创新者的知识。自1978年开始O’Reilly一直都是发展前沿的见证者和推动者。超级极客正在开创未来，我们关注着真正重要的技术趋势，通过放大那些“微弱的信号”来刺激社会对新科技的采用。作为技术社区中活跃的参与者，O’Reilly的发展充满着对创新的倡导、创造和发扬光大。

作为出版商O’Reilly为软件开发人员带来革命性的“动物书”，创造了第一个商业网站（GNN），组织开放源代码峰会以至于开源软件运动以此命名，通过创立Make杂志成为DIY革命的主要先锋，公司一如既往地用各种方式和渠道连接人们和他们所需要的信息。O’Reilly的会议和峰会聚集了超级极客和高瞻远瞩的商业领袖，共同描绘将开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通计算机用户。无论是通过印刷书籍、在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的信念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位少有的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

本书的翻译团队包括鲍央舟、沈欢星和张玳，三者均为软件行业从业人员。在半年中抽取了大量空余时间投入本书的翻译和审校工作。在翻译的过程中，遇到了众多数学、统计学、心理学的专业术语，在一些朋友的帮助下，才得以顺利完成。在这里想感谢所有帮助过我们的朋友，也感谢我的公司Out Softing对我工作的支持。如有疏漏和不足之处，希望广大读者批评指正。

鲍央舟

前　　言

MMR疫苗会引发孤独症吗？电视里的暴力镜头会使孩子们更暴力吗？某些编程语言比其他一些更好吗？人们每天都会争论这些问题。要认真地回答前两个问题必须依靠科学方法：小心地收集证据，公平地评估效果。然而，迄今为止，很少有人试图用这样的技巧来回答第三个问题。当说到计算机相关工作的时候，边喝啤酒边说出来的有关华沙创业公司的轶事，通常就是大部分程序员所期望的所有“证据”了。

这种情况正在改变，部分归功于本书撰稿人的工作。本书作者和他们的同事从不同领域获取数据，诸如数据挖掘、认知心理学以及社会学……他们正在创造一种软件工程的循证方法。通过从无数初始材料中搜集证据并分析结果，他们正在为一些软件工程的恼人问题带来新的光明。大部分程序员在他们的第一份工作中会如何出错？测试驱动开发会产生更好的代码吗？结对编程或代码审查又如何？可能在发布之前预测一段代码中缺陷的大概数目吗？如果能的话，怎么做？

这本书中的论文会提供一些问题的解答，并解释为什么其他问题仍然没有答案。同样重要的是，它们会告诉你如何用定量和定性的方法自己找到并评估证据。每个程序员都是独特的，也没有任何两个程序是完全相同的，但如果你仔细、耐心、开明的话，就能说服他们说出秘密。

我们希望本书中的问题和解答能改变你对软件开发的想法。我们也希望这些论文能说服你，在下次有人声称C和Java中这样放置大括号比那样放置更好的时候，你会说“请举证”。和《代码之美》[①]一样（Andy和Greg与O’Reilly的上次合作），作者的所有版税会捐赠于有关机构。

本书的结构

本书中的每一章节都来自不同的撰稿人或团队。顺序并不重要，但是我们把一些关于研究、有效性和其他具有高层意义的章节放在开头。我们觉得在读过第一部分后，读者会掌握一定背景知识，从而能更好地理解第二部分的内容。

第一部分包含以下内容：

第1章　探寻有力的证据（Tim Menzies和Forrest Shull著）

第2章　可信度，为什么我坚决要求确信的证据（Lutz Prechelt和Marian Petre著）

---

① 该书由机械工业出版社于2009年出版。——编者注

第3章 我们能从系统性评审中学到什么（Barbara Kitchenham著）
第4章 用定性研究方法来理解软件工程学（Andrew Ko 著）
第5章 在实践中学习成长：软件工程实验室中的质量改进范式（Victor R. Basili 著）
第6章 性格、智力和专业技能对软件开发的影响（Jo E. Hannay著）
第7章 为什么学编程这么难（Mark Guzdial著）
第8章 超越代码行：我们还需要其他的复杂度指标吗（Israel Herraiz和Ahmed E. Hassan著）
第二部分包含以下内容：
第9章 自动故障预报系统实例一则（Elaine J. Weyuker和Thomas J. Ostrand著）
第10章 架构设计的程度和时机（Barry Boehm著）
第11章 康威推论（Christian Bird著）
第12章 测试驱动开发的效果如何（Burak Turhan、Lucas Layman、Madeline Diep、Hakan Erdogmus和Forrest Shull著）
第13章 为何计算机科学领域的女性不多（Michele A. Whitecraft和Wendy M. Williams著）
第14章 两个关于编程语言的比较（Lutz Prechelt著）
第15章 质量之战：开源软件对战专有软件（Diomidis Spinellis著）
第16章 码语者（Robert DeLine著）
第17章 结对编程（Laurie Williams著）
第18章 现代化代码审查（Jason Cohen著）
第19章 公共办公室还是私人办公室（Jorge Aranda著）
第20章 识别及管理全球性软件开发中的依赖关系（Marcelo Cataldo著）
第21章 模块化的效果如何（Neil Thomas和Gail Murphy著）
第22章 设计模式的证据（Walter Tichy著）
第23章 循证故障预测（Nachiappan Nagappan和Thomas Ball著）
第24章 采集缺陷报告的艺术（Rahul Premraj和Thomas Zimmermann著）
第25章 软件的缺陷都从哪儿来（Dewayne Perry著）
第26章 新手专家：软件行业的应届毕业生们（Andrew Begel和Beth Simon著）
第27章 挖掘你自己的证据（Kim Sebastian Herzig和Andreas Zeller著）
第28章 正当使用“复制–粘贴”大法（Michael Godfrey和Cory Kapser著）
第29章 你的API有多好用（Steven Clarke著）
第30章 “10倍”意味着什么？编程生产力的差距测量（Steve McConnell著）

本书的排版规范

本书使用的排版规范如下所示。

- 楷体

用于表示新的术语。

- 等宽字体

表示程序片段，也在段落中表示程序中使用的变量、函数名、命令行实用工具、环境变量、语句和关键字等元素。

- 等宽加粗

这种字体表示用户需要手动输入的命令或者相应的文本。

- 等宽斜体

用户需要根据自己所提供的值或由上下文所确定的值进行更改的部分。

Safari®在线图书



Safari在线图书是应需而变的数字图书馆。它能够让你非常轻松地搜索7500多种技术性和创新性参考书以及视频，以便快速地找到需要的答案。

订阅后就可以访问在线图书馆内的所有页面和视频。可以在手机或其他移动设备上阅读。还能在新书上市之前抢先阅读，也能够看到还在创作中的书稿并向作者反馈意见。复制粘贴代码示例、放入收藏夹、下载部分章节、标记关键点、做笔记甚至打印页面等有用的功能可以节省大量时间。这本书也在其中。

欲访问本书英文版的电子版，或者由O’Reilly或其他出版社出版的相关图书，请到http://my.safaribook- sonline.com免费注册。

使用代码范例

让本书助你一臂之力。也许你要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布O’Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

我们非常希望你能标明出处，但并不强求。出处一般含有书名、作者、出版商和ISBN，例如：“*Making Software :What Really Works, and Why We Believe it* by Andy Oram and Greg Wilson. Copyright 2011 O’Reilly，Media, Inc., 978-0-596-80832-7”。

如果有关于使用代码的未尽事宜，可以随时与我们取得联系：permissions@oreilly.com。

我们的联系方式

请把对本书的评论和问题发给出版社。

美国：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

O’Reilly的每一本书都有专属网页，你可以在那儿找到关于本书的相关信息，包括勘误表、示例代码以及其他的信息。本书的网站地址是：

http://www.oreilly.com/catalog/9780596808327/

中文书：

http://www.oreilly.com.cn/book.php?bn=index.php?func=book&isbn=9787115270443

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资源中心和网络，请访问以下网站：

http://www.oreilly.com

http://www.oreilly.com.cn

目　　录

Part 1

第一部分

搜寻和使用证据的一般原则

当研究人员调查程序员的行为时，他们的方法必须变得比以往更加精妙，因为这个棘手的领域涉及人的因素。首先，我们必须知道我们能学多少，以及学到什么程度时我们必须接受知识范围的局限性。因此，这一节会讨论一些原则，帮助我们找到可信任的软件工程数据，并小心适当地运用于新情况。章节中会触及一些数据的运用，但关注的焦点还是在于更宽泛的主题，以期读者能利用这些知识更好地解决软件工程中其他的争议话题。

第1章

探寻有力的证据

Tim Menzies
Forrest Shull

是什么使证据精确、合理、有用并有信服力？这正是本章作者在过去20年中所探寻的问题。我们两人在实证软件工程这个话题上发布的文章数目加起来都超过了200篇，组织了无数次专家小组讨论、讲习班、会议以及杂志特刊的发布。

在这篇文章中，我们想要稍微反思一下。我们回顾了迄今为止软件工程的进程，并自问："这到底有什么好处？""我们该走向哪里？"以下是我们的发现。

- 对有力证据的探寻比我们开始想的要复杂得多。
- 软件工程研究人员需要更多地工作在一个团体中，分享更多的证据。
- 我们需要承认证据多少是与背景相关的。那些使我们信服的证据并不一定使你们信服。因此，我们需要准备好，一旦听众改变，就需要重新解释和重新分析。

1.1 起步阶段

几十年前，当被问起什么是我们认为的"完美的证据"时，我们会提到下列这些特征。

- 研究的精确性

  许多软件工程的研究包含了人为因素。因为很多软件技术的有效性很大程度上依赖于使用它们的人①。但是处理人的可变性是很具挑战的。通过精妙设计来最小化这些混杂效应的研究能使别的研究者倍感羡慕。比如，Basili和Selby的一个研究[3]使用了部分析因设计②，在这个设计中，每个开发人员使用的都是有待审查的技术，并且所有技术都被使用在实验中的程序片段。

- 统计的强度

  随着数学复杂度的增长，对统计学显著性结果的关注也随之增长。以此，研究者能对他们的理论在现实世界中的作用有一些信心，即使在随机背景的干扰下也能识别出来。

① 在研究了161个项目之后，Boehm等人于2000年发现最好的项目人员的产出率是最差的3.5倍。

② Fractional Factonial Design，选择析因实验（Factonial Experiment）的一部分来研究。——编者注

- 结果的可重复性

  如果结果可以在许多不同背景下被重复发现，也就是说，结果不止局限于一种背景或一组实验条件，这样的结果会更有说服力！在其他学科中，复制增强信心，出于这个原因，很多人花了很多精力，试图使软件工程实验可简单地被其他研究者在其他背景下重复运行[4]。举一个可重复性的例子，Turhan证明了在其他站点习得的软件缺陷预报器可以成功运用于新站点[31]。

1.2 当今证据的状态

回顾以往，我们现在才知道当时对有力证据的定义是多么幼稚。精确、统计性强、重复证据，这些都最终被证明比我们想的要难找得多。另外，它们并不能满足与研究更相关的目标。

1.2.1 精确性研究的挑战

我们发现精确的研究可能对那些接受过足够科研培训的人来说很有说服力，但对那些普通从业者却很难解释。因为这样的研究通常会有所限制和简化，从而使得研究背景不是以代表真实的开发环境。比如，Basili和Selby的研究仅针对“迷你”问题运用了研究中的技巧，该问题只在虚拟环境中有不超过400行的代码。这项研究经常被引用，虽然它是被经常复制的对象，但似乎没有一项复制使用了更大规模的或者更具代表性的应用[4]。虽然这项精确的研究对我们理解铲除代码缺陷的不同方法的优劣有重大的贡献，但如果我们在这一主题上的大部分思维来自相对小的代码段，这并不理想。

1.2.2 统计强度的挑战

什么构成了对现实世界问题的“强有力”统计数字？对于这个问题的共识出奇得少。首先，有一个关于外部有效性的问题：经过充分测试的度量方法是否能反映我们所关注的真实世界的现象。比如，Foss等人证明了常用估计工作量的评估方法根本就是有问题的[14]。更糟糕的是，他们指出，没有单个的解决方法：

> “寻找‘圣杯’是徒劳的：单个的、简单易用的、普遍适用的、使用起来可以方便地（和不同方法）做比较的度量方法，根本不存在。”

此外，不同的作者使用完全不同的统计分析法，没有一个方法被广泛地接受为“最强有力的”方法。

- ❑ Demsar记录了在一个著名国际会议上看到的巨大数量的统计方法，那个会议专注的是从数据中学习经验教训[11]。
- ❑ Cohen讨论了使用标准统计假设测试来做科学结论的方法。他尖刻地描述这种测试是“有力但毫无结果地耙平了智力成果，剩不下任何可生长的科学种子”。[10]

为了支持Cohen的论题，我们再列举一个有益的教训。在营销领域中，Armstrong[1]使用显著

性测试复查了一个研究，该研究的结论指出，从多样本来源产生的估计并不比单样本来源产生的估计要好。他通过列出了31项研究结果推翻了这个结论。在这31项研究中，多样本来源预测的效果始终超过单样本来源预测3.4%～23.4%（平均12.5%）。在Armstrong的每项调查中，这些超越都与显著性测试所预计的结果正好相反。

基于这些发现，我们改变了对统计分析的看法。现在我们会尽可能使用简洁的可视化方法来给出论点，并且把统计的显著性测试降级至“合理性测试”的角色，来验证从可视化方法中做出的结论）。①

1.2.3　结果可复制性的挑战

可复制性已被证明是一个很难捉摸的目标。在某些主题上，很少有证据证明一个项目的结果已经或者可以被推而广之。

- Zimmermann研究了629对软件开发项目[34]。在只有4%的情况中，从一个项目中习得的缺陷预测模型对另一个有用。
- Kitchenham等人的一项调查声称第一个项目的数据对第二个项目的工作量估计是有用的[18]。但他们发现现有的证据没有说服力，甚至是自相矛盾的。

在其他主题上，我们可以发现特定现象对不同环境都奏效的证据。比如，像软件检查这样成熟的技术可以找到软件产品中大量的现存缺陷[27]。然而，如果一个新的研究提出了反面证据，那也很难确定新的（或旧的）研究是有缺陷的，或者研究其实只是在一个特殊的环境下运行的。鉴于软件开发环境的多样性，两个结论通常都是有可能的。

事实上，虽然直觉上我们可能愿意相信可复制的结果，但要么关于特定的软件工程问题的研究就很少，要么它们并不全面。

作为对研究匮乏的例证，Menzies研究了不同组织提出的100个软件质量保证方法（如IEEE1017和NASA IV&V内标），发现没有实验能证明任何一个方法比别的方法更划算[21]。

现有研究不全面的例子包括如下所列。

- Zannier等人随机抽取了所有ICSE发布文章中的5%作为研究对象[33]，ICSE是自评为排名第一的软件工程大会。他们发现那些自称为“实证”的文章中，只有极少数（2%）比较了不同研究者的方法。
- Neto等人汇报了一项，针对基于模型测试（MBT）方法的文献的调查结果[23]。他们发现有85篇文章描述了71个独特的MBT方法，却只有很少数的研究有实验的部分。这显示了研究人员的总体趋势是持续创新和汇报新的方法，而不是理解可比的现有方法中的实际利益。

为了看看这些文章反应的到底是一种个别情况还是一种更普遍的趋势，我们审阅了所有PROMISE②大会上做过的关于可重复软件工程实验的演讲。从2005年起，在PROMISE大会上。

① 但我们仍然每月驳回大约两篇申请杂志发表的文章，因为他们只汇报平均值结果，没有任何平均值附近多方差的统计上或可视化的表达。最起码，我们推荐Mann-Whitney或者Wilcoxon测试（分别针对非配对结果和配对结果）来证明明显不同的结果真的可能是不同的。

② 查看http://promisedata.org。

- 一共有68个演讲，其中48个尝试了对旧数据的新研究或者以Zannier[33]的形式做了汇报，即一个对特定项目生效的新方法。
- 9篇文章对之前研究结果的有效性提出了质疑（如 Menzies 的报告[22]）。
- 4篇文章主张软件工程模式的通用性是不可能的（如 Briand 的报告[7]）。
- 只有很少的研究者（68个演讲中的7个）汇报了从一个项目到另外项目的推广：
  - 4篇文章汇报了从一个项目中习得的软件质量预警器在一个新的项目中成功使用（如Weyuker等人[32]和Tosun等人[30]的报告）；
  - 3篇文章举了特定案例说明通用性是可行的（如Boehm的报告[5]）。

这些发现稍微引起了我们的一些警惕，我们与欧美领先的实证软件工程研究者进行了讨论。我们的谈话可以总结为以下几点。

- Vic Basili在过去30年中是实证软件工程（SE）的先驱。在断言现在的实证软件工程已经比20世纪80年代健康许多后，他承认了两点。第一，迄今为止的研究结果都是不全面的。第二，几乎没有例子能举出有多个项目可用的方法[2]。
- David Budgen和Barbara Kitchenham一起，是欧洲提倡“循证软件工程”（EBSE）的领军者。在EBSE中，软件工程师的实践应该基于有文献充分支持的方法。Budgen和Kitchenham试图探明：“循证软件工程是否在实践和政策上已经足够成熟？”他们的回答是“不，还不是”：软件工程领域需要大幅调整，只有在那之后，我们才能证明研究结果是可以在不同项目中复制的[8]。它们主张软件工程中不同的报告标准，特别是使用“结构式摘要”来简化对SE文献的大规模分析。

1.3 我们可以相信的改变

至今，我们还未实现对精确、统计充分、可复制证据的梦想。而且，即使找到了符合单个特征的证据，也没有我们想象中的有影响力。也许，我们需要重新审视对“有力证据”的定义了。根据前几节的反馈，是否有更实际的对有力证据的定义，以此激励研究者？

（如果谦虚地说）更可行的定义是：有力的证据*激发改变*。我们猜想，这本书中的很多作者都是因为看到真实软件开发的第一手问题和困难，才开始了对有力证据的搜寻。这些研究者寻找的“圣杯”应该是那些能够创造真实世界进步的研究结果。

为了激发改变，一些有影响力的读者不得不去相信证据。有一种处理不够严格的经验报告或案例分析的方法，是对每件证据赋予一个“信心指数”。这个指数试图反应每篇报告在一定信心范围内的位置，从传闻性的证据，或者说有问题的证据，到非常可信的证据。Kitchenham建议对系统性评论和软件工程研究使用新的流程，而这样的评估正是该流程的必要部分[17]。在Feldmann的一篇文章中，也可以找到一种帮助从业者交流信心指数的简单刻度[13]。

生成真正令人信服的证据也需要对传播结果的途径做出改变。也许，科技出版物不再是传播证据的唯一技术。现在的出版物存在很多问题，其中之一就是，这些出版物通常不提供原始数据，其他研究者就不能重新分析、重新解译并验证。另外，虽然出版物的作者非常努力地对背景做出

详尽的描述，但也不可能列出所有相关因素。

更有希望的一种做法是，创造软件工程数据知识库，用来存储跨环境的研究成果（至少是跨项目的），然后可以从这个仓库的数据中展开分析。如下所示。

- 内布拉斯加大学的软件构件基础设施研究（SIR）中心[①]

  这个知识库存储了软件相关的构件，研究者可以把它用在有程序分析和软件测试技术严格控制的实验法中。教师也可以用它在可控实验中来训练学生。这个知识库包含了很多Java和C的不同版本软件系统，也包含了支持构件，如测试套件、缺陷数据和脚本。这个知识库中的构件已经在上百个出版物中被使用。

- NASA软件工程实验室（SEL）

  NASA实验室（详见第5章）是一个巨大的成功。它有广泛的影响力，它的数据和结果也持续被引用。有一个涉及背景的重要经验：实验室的领导者要求想用他们数据的研究者花时间做实验，从而正确理解他们的研究背景，不曲解他们的分析结果。

- CeBASE

  这是一个数据和经验的知识库，由NSF资助，用于与其他研究者共享并重新分析。我们利用了SEL的经验，探索把数据用较少的开销置于不同背景的方法，如用一套丰富的元数据标记所有的数据集，元数据描述了数据的出处。这些数据经验回过头来成为另一个“经验教训”知识库的支柱，这个知识库由美国国防部的国防采办大学维护，可以让最终用户指定他们自己的背景参数，比如项目的大小、重要性或者领域，以此找到在相似环境中被证实的实践。

- PROMISE项目

  另一个活跃的软件工程知识库来自于PROMISE项目，这个项目在本章的前面也被引用过。PROMISE探寻可重复的软件工程经验。这个项目分为三个部分。

  - 在线公共域数据知识库。在写此文时，该知识库包含91了个数据集。其中一半关于缺陷预警，剩余的数据集探索了工作量预估、基于模型的软件工程、SE数据的文本挖掘以及其他问题。
  - 每年一次的年会，强烈鼓励作者不仅发布文章，也在使用数据知识库做出他们的结论后，为知识库做贡献。
  - 特别期刊，发表年会中最好的文章。[19]

知识库在提供链接至数据贡献者时特别有用。我们必须承认，数据很少独立存在。作者应该把回答问题以及与试图解读他们工作的人对话视为常态，而非反常情况。这种接触对理解得出数据的背景很重要。也能帮助用户找到与他们的问题和他们的环境最相关的数据。巴尔的摩市马里兰大学（UMBC）的Gunes Koru在PROMISE的一些数据集中找到了一个系统错误。他通过PROMISE知识库附属的博客软件张贴了这个错误。然后，通过PROMISE的博客[②]，他联系到了原始数据的制造者，这些制造者对这些错误的原因以及避免方法提供了大量的评论。

---

① 查看http://sir.unl.edu。

② 查看 http://promisedata.org/?p=30#comments。

1.4　背景的影响

PROMISE这样的知识库虽然是找到有推动性和有说服力证据的必要因素，但它们也只是解决方法的一部分。我们在之前章节中描述了为什么证据的收集总是与上下文相关的。最近我们开始认识到，解读证据也是与上下文相关的，甚至是与读者相关的。为了更好地认识这个问题，我们需要离题一会，去讨论一点理论。

许多软件工程的问题存在于一个解空间内，这个解空间就像随意扔在地板上的地毯一般任意扭曲。想象一个蚂蚁正在搜寻这块地毯的高峰和低谷，试图找到最低点，比如在这个点上，软件开发的工作量和缺陷的数量最少。

如果问题足够复杂（软件设计和软件流程的决定也确实非常复杂），就不存在找到最佳解决方案的最佳途径。也就是说，蚂蚁可能被困在错误的低谷中，自认为是最低其实却不是（比如，一些山脊阻碍了它的视线，使之看不见相邻的更低谷）。

优化算法和人工智能算法使用各种试探性方法来搜索这个选择空间。一种方法是根据特定观众的目标对问题的上下文建模，然后把搜索引导至一个特定的方向。想象那只蚂蚁在一根皮带上，而这根皮带向着目标方法被轻轻地拉动。

现在，令人吃惊的来了。这些试探性搜索方法虽然有用，但却对研究结果施加了一个偏差。如果你改变上下文，也改变了偏差，那这些算法就会找到不同的“最佳”方案。例如，在对软件过程模型的人工智能搜索实验中，Green等人使用了两种不同的目标函数[15]。一个目标代表了安全性至关重要的政府项目，试图同时减少开发工作量和软件交付的缺陷数。另一个目标代表了更标准的商业情况，急于发布软件至市场，并一直努力不要插入过多缺陷。此研究用人工智能优化器搜索了4个不同项目。每次搜索在每个目标下重复。一个惊人的结果是，*一个目标产出的建议通常在另一个目标下会被否决*。比如，使用一个目标的人工智能搜索推荐*增加*交付前的时间，而另一个则建议*减少*时间。

对任何试图找到证据来说服人们改变现有软件开发流程和工具的人来说，这个结果都有重要意义。我们需要*根据观众调整证据*，而不是假设所有证据都能说服所有人。站在台上像变戏法似的拿出看似动人的证据并不足够。即使是从精确研究中取得的有统计强度的、可复制的证据，如果证据与观众问题不相关，那也不能激发任何改变。换句话说，观众可能会自问：“这到底有什么好处？”而我们需要尊重他们的“业务偏差”。

1.5　展望未来

虽然软件工程研究已经进行了几十年，但是至今我们只看到了极少的有力证据，能引导软件项目运行方式的改变。我们推测这是由于背景的问题：研究者制造了关于A的证据，而观众却关注B、C、D等。我们推荐研究者在搜寻软件工程证据时多一点谦卑，至少和现在持平，并愿意结识并倾听软件从业者，他们可以帮助我们更好地领会B、C、D究竟是什么。我觉得我们的领域需要在至少一段时间内，停止搜寻对所有项目所有情况都适用的结果的证据，找到有影响力的局部

结果已经足够有挑战性了。

Endres和Rombach提出了一个，关于如何构建软件和系统工程知识的观点[12]。

- 在特定环境下对实际开发工作进行观察，这在任何时候都能进行。（“观察”在这里的定义同时包含铁的事实和主观印象。）
- 重复的观察会引出一些规则，帮助理解事情将可能会如何发生。
- 规则可以被理论所解释，解释为什么这些事件发生了。

鉴于当前经验性研究处理问题的复杂性，我们不应该急于进行理论建设。一个更多产的知识构建模型基于我们现在看到的所有数据和研究，是一种“观察”和“规则”的双层方法（使用Endres的术语），由我们之前描述的知识库来支撑。

对第一层来说，研究者会对局部观众的目标建模，然后收集关注这些目标的证据。已有的现代化数据挖掘工具[①]，可以帮助工程师们学习局部的经验教训。

在第二层中，我们把不同项目和不同背景的结论抽象出来。现在可能必须满足于只在一个领域中抽象出重要因素或基本原则，而不是提供“唯一”的解决方法对所有不同背景的子集都适用。

比如，Hall等人试图回答什么能激励软件开发者。[16]他们查看了92个实验过此问题的研究，每个都有不同的背景。在试着弄清楚这些研究和它们的不同研究结果时，研究者在寻找看似能激励开发人员的因素，即使不太可能量化这些因素的贡献。于是，伴随着一定的可信度，在多个研究中所找到的有作用的激励因素被包含在这个模式中。

最终结果不是一个预言性的模型，如X因素是Y因素重要性的两倍。相反，它是一个重要因素的列表，管理者可以在自己的环境中用它来确保他们没有忽视或忘记某些东西。也许，在许多问题上，我们能做到的最好情况是，提供需要考虑的因素来装备那些带着问题的从业者，让他们找到自己问题的解决方案。

要做到这点，我们需要扩大第一层观察所能接受的“证据”的定义。一些研究者一直争论说，软件工程研究存在对定量数据和研究的偏好，而定量的工作也可以同样严格并提供对相关问题的有用回答[26]。构建更强大证据集合的开始，是真正拓展可接受证据的定义，以此混合定性和定量的资源。也就是说，更多关于技术为什么能行或者为什么不行的文本或图片数据，以及技术影响力多少的定量数据。

然而，真正有影响力的证据实体走得更远，并且接收完全不同种类的证据。不止是试图找到具有统计意义的研究，也包含能提供更多关于技术实践应用信息的调查经验汇报。这种经验汇报目前被低估了，因为它们被认为不如现有的文献严格。比如，不总能确保各方面利益都被精确衡量，不能保证混淆的因素已经排除，或者不能证明流程一致性因素已经避免。然而，这些报告应该是任何软件开发技术“证据线索”的一个显而易见的部分。

正如Andres把“观察”定义地足够广泛来包含主观印象，如果我们不根据他们的标记去创建无根据的规则，即使不那么严格的输入模式也可以帮助我们找到有效的结论。对这些证据源赋予信心指数非常重要，这样，方法论的问题就可以被凸显出来。（当然，即使最严格的研究也不会

---

① Weka：http://www.cs.waikato.ac.nz/ml/weka; R: http://www.r-project.org; YALE: http://rapid-i.com/。

完全没有任何方法论的问题。）

经验汇报可以通过证明实践约束下达成的结果不总是与我们对给定技术的期望相匹配，从而使研究有根有据。同样重要的是，他们对于需要如何剪裁或修改技术来应对日常软件开发的实践约束提供了见解。在技术转移领域，我们经常发现，一个描述技术在实践中正面经验的案例分析，对从业者来说，比大量额外的研究报告更有价值。如此令人信服的证据体，也能通过提供对不同用户有用的证据来帮助引导改变。Rogers提出了一个经常被引用的模型，用钟形曲线刻画了一些创新产品（如研究结果）的消费者：最左边的尾巴包含了创新者，钟形的突出部分代表了大多数人，他们随着这个创新想法逐渐流行而采纳，最右边的尾巴包含了抵触改变的落后者[24]。了解自己观众的研究者能从真正健壮的数据集中选择适当的子集来构建他们的案例。

- ❑ 早期采纳者可能找到的是包含相对低可信度的可行性研究的一个小集合，或者一到两个“精确”研究，这已经足够使他们采纳改变。特别是当这些研究的背景显示出证据是在一个与他们相似的环境中收集的。
- ❑ 大部分采纳者需要看到不同上下文中的多样研究，才会相信研究的想法是有价值的，被证实不止在一个合适的环境中可行，并且已经开始成为被部分接受的行事方式。

落后者或者后期采纳者可能需要压倒性的证据，这可能包括大量的高可信度研究，以及在大范围不同背景下得到的有益结果。

一些混合这些不同类型证据的方法已经被提出[29]。但最重要的是，定性报告和定量数据之间的相互影响。我们经常看到从业者在混合不同数据类型的数据集后收到更好的效果。例如，一个同时包含硬数据和真实团队正面经验的丰富集合，帮助了软件检查技术在不同NASA中心的散播[28]。

从良好设计的经验性研究中得出的证据可以帮助结果达到统计意义，但实施可能仍然不切实际，或者不能及时回答新问题（特别在技术快速改变的领域，如软件工程）。实际运用中的证据可能对益处更有说服力，但是通常不太严格。经常需要同时结合两种证据，相互巩固，才会最具说服力。

1.6 参考文献

[1] [Armstrong 2007] Armstrong, J.S. 2007. Significance tests harm progress in forecasting. *International Journal of Forecasting* 23(2): 321-327.

[2] [Basili 2009] Basili, V. 2009. Personal communication, September.

[3] [Basili and Selby 1987] Basili, V., and R. Selby. 1987. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering* 13(12): 1278-1296.

[4] [Basili et al. 1999] Basili, V.R., F. Shull, and F. Lanubile. 1999. Building Knowledge Through Families of Experiments. *IEEE Transactions on Software Engineering* 25(4): 456-473.

[5] [Boehm 2009] Boehm, B. 2009. Future Challenges for Software Data Collection and Analysis. Keynote address presented at the International Conference on Predictor Models in Software Engineering (PROMISE09), May 18-19, in Vancouver, Canada.

[6] [Boehm et al. 2000] Boehm, B.W., C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. 2000. *Software Cost Estimation with Cocomo II.* Upper Saddle River, NJ: Prentice Hall PTR.

[7] [Briand 2006] Briand, L. 2006. Predictive Models in Software Engineering: State-of-the-Art, Needs, and Changes. Keynote address, PROMISE workshop, IEEE ICSM, September 24, in Ottawa, Canada.

[8] [Budgen et al. 2009] Budgen, D., B. Kitchenham, and P. Brereton. 2009. Is Evidence Based Software Engineering Mature Enough for Practice & Policy? Paper presented at the 33rd Annual IEEE Software Engineering Workshop (SEW-33), October 13-15, in Skövde, Sweden.

[9] [Carver et al. 2008] Carver, J., Juristo, N., Shull, F., and Vegas, S. 2008. The Role of Replications in Empirical Software Engineering. *Empirical Software Engineering: An International Journal* 13(2): 211-218.

[10] [Cohen 1988] Cohen, J. 1988. The earth is round (p < .05). *American Psychologist* 49: 997-1003.

[11] [Demsar 2006] Demsar, J. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* 7: 1-30.

[12] [Endres and Rombach 2003] Endres, A., and H.D. Rombach. 2003. *A Handbook of Software and Systems Engineering*. Boston: Addison-Wesley.

[13] [Feldmann et al. 2006] Feldmann, R., F. Shull, and M. Shaw. 2006. Building Decision Supportin an Imperfect World. *Proc. ACM/IEEE International Symposium on Empirical Software Engineering* 2: 33-35.

[14] [Foss et al. 2003] Foss, T., E. Stensrud, B. Kitchenham, and I. Myrtveit. 2003. A Simulation Study of the Model Evaluation Criterion MMRE. *IEEE Transactions on Software Engineering* 29(11): 985-995.

[15] [Green et al. 2009] Green, P., T. Menzies, S. Williams, and O. El-waras. 2009. Understanding the Value of Software Engineering Technologies. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering:* 52-61. Also available at *http://menzies.us/pdf/ 09value.pdf.*

[16] [Hall et al. 2008] Hall, T., H. Sharp, S. Beecham, N. Baddoo, and H. Robinson. 2008. What Do We Know About Developer Motivation? *IEEE Software* 25(4): 92-94.

[17] [Kitchenham 2004] Kitchenham, B. 2004. Procedures for undertaking systematic reviews. Technical Report TR/SE-0401, Department of Computer Science, Keele University, and National ICT, Australia Ltd.

[18] [Kitchenham et al. 2007] Kitchenham, B., E. Mendes, and G. H. Travassos. 2007. Cross VersusWithin-Company Cost Estimation Studies: A Systematic Review. *IEEE Trans. Software Eng.* 33(5): 316-329.

[19] [Menzies 2008] Menzies, T. 2008. Editorial, special issue, repeatable experiments in software engineering. *Empirical Software Engineering* 13(5): 469-471.

[20] [Menzies and Marcus 2008] Menzies, T., and A. Marcus. 2008. Automated Severity Assessment of Software Defect Reports. Paper presented at IEEE International Conference on Software Maintenance, September 28-October 4, in Beijing, China.

[21] [Menzies et al. 2008] Menzies, T., M. Benson, K. Costello, C. Moats, M. Northey, and J. Richardson. 2008. Learning Better IV & V Practices. *Innovations in Systems and Software Engineering* 4(2): 169-183. Also available at *http://menzies.us/pdf/07ivv.pdf.*

[22] [Menzies et al. 2009] Menzies, T., O. El-waras, J. Hihn, and B. Boehm. 2009. Can We Build Software Faster and Better and Cheaper? Paper presented at the International Conference on Predictor Models in Software Engineering (PROMISE09), May 18-19, in Vancouver, Canada. Available at *http://promisedata.org/pdf/ 2009/01_Menzies.pdf.*

[23] [Neto et al. 2008] Neto, A., R. Subramanyan, M. Vieira, G.H. Travassos, and F. Shull. 2008. Improving Evidence About Software Technologies: A Look at Model-Based Testing. *IEEE Software* 25(3): 10-13.

[24] [Rogers 1962] Rogers, E.M. 1962. *Diffusion of innovations*. New York: Free Press.

[25] [Runeson et al. 2006] Runeson, P., C. Andersson, T. Thelin, A. Andrews, and T. Berling. 2006. What do we know about defect detection methods? *IEEE Software* 23(3): 82-90.

[26] [Seaman 2007] Seaman, C. 2007. Qualitative Methods. In *Guide to Advanced Empirical Software Engineering*, ed. F. Shull, J. Singer, and D. I. K. Sjøberg, 35-62. London: Springer.

[27] [Shull 2002] Shull, F., V.R. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M.V. Zelkowitz. 2002. What We Have Learned About Fighting Defects. Proc. *IEEE International Symposium on Software Metrics (METRICS02)*: 249-258.

[28] [Shull and Seaman 2008] Shull, F., and C. Seaman. 2008. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software* 24(7): 88-90.

[29] [Shull et al. 2001] Shull, F., J. Carver, and G.H. Travassos. 2001. An Empirical Methodology for Introducing Software Processes. *Proc. Joint European Software Engineering Conference and ACMSIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE):* 288-296.

[30] [Tosun et al. 2009] Tosun, A., A. Bener, and B. Turhan. 2009. Practical Considerations of Deploying AI in Defect Prediction: A Case Study within the Turkish Telecommunication Industry. Paper presented at the International Conference on Predictor Models in Software Engineering (PROMISE09), May 18-19, in Vancouver, Canada. Available at *http://promisedata .org/pdf/2009/09_Tosun.pdf*.

[31] [Turham et al. 2009] Turhan, B., T. Menzies, A.B. Bener, and J. Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14(5):540-557.

[32] [Weyuker et al. 2008] Weyuker, E.J., T.J. Ostrand, and R.M. Bell. 2008. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering* 13(5): 539-559.

[33] [Zannier et al. 2006] Zannier, C., G. Melnik, and F. Maurer. 2006. On the Success of Empirical Studies in the International Conference on Software Engineering. *Proceedings of the 28th international conference on software engineering*: 341-350.

[34] [Zimmerman 2009] Zimmermann, T., N. Nagappan, H. Gall, E. Giger, and B. Murphy. 2009. Cross-Project Defect Prediction. *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*: 91-100.

第2章

可信度，为什么我坚决要求确信的证据

Lutz Prechelt
Marian Petre

作为软件工程师，我们对什么起作用、什么不起作用或者不太起作用都有各自的意见。我们因此共享实践故事和经验，它们最终化作文化知识和各种常识。但问题是，“地球人都知道的东西”常常是错的。虽然我们不断收集信息，但是也许并未细致准确地评估并整合那些信息，甚至都没有必要的手段来这么做。

2.1 软件工程中的证据是如何发现的

在侦探片中，新证据出现的时刻通常是最有趣的：某个人提供了之前不为人知的事实，然后聪明的侦探调整了他的犯罪推理。有时，新的证据为我们的世界观加入了一个重要维度（“噢，原来他预谋了所有这些事情！”）；有时，它只是让我们更加坚信自己的想法（“她确实不像她说的那么仔细”）；有时，它会颠覆我们之前认为很可靠的东西（“天哪！我一直以为他比她晚到了一小时！”）。故事情节取决于侦探的判断性思维能力。小说里的卓越侦探们会把所有零星的证据列入考虑范围内，不把所有证据结合成一个整体不罢手，然后不断地测试他们的犯罪推理，一旦看到新证据就会相应调整。然而一个差的侦探则会执著于确认式偏见[①]，即使他们不能说明证据中的剩余疑点，也会坚守自己的推理。

软件开发也类似：证据随着时间的推移不断出现，而工程的质量则取决于工程师的判断能力。如果我们屈服于确认式偏见，那么我们就会把大部分注意力集中到巩固我们自己观点的证据中。如果我们能更警觉一些，有时就会找到新的信息，提醒我们之前忽视了某些方面，应该把它放入全局考虑。也许，我们甚至能发现某个深入人心的信念其实是与事实相矛盾的。

相关的新信息从哪里来呢？有以下多种渠道。

① 确认式偏见（confirmatory bias）：指人们倾向于采纳支持自己的设想或者预期的证据，而不管这些证据是否真实。——译者注

- 经验

  软件工程师通过不断使用技术和方法、参与项目来学习新东西。

- 他人

  软件工程师与认识的人交谈，并倾听他们尊敬的人给的建议。

- 反思

  出色的软件工程师会努力思考他们的所作所为，思考哪些起效、哪些不起效和为什么。

- 阅读

  书面材料，无论是非正式的（如高质量的博客文章）还是正式的（如科技文章）都传输了其他各方的见解。

- 科学（或类科学）探索

  软件工程师进行软件实验、着手用户研究并对各种选择开展系统性比较。

并不是所有人都认为软件工程掌握了足够的证据。已经有一些人呼吁“循证软件工程”（evidence-based software engineering，如Finkelstein[3]和Kitchenham[6]等人的文章）的方法，即整合实证研究的结果提供技能、习惯和基础，以便支持软件工程的决策。我们通常会拿软件工程与循证医学及其对严格临床证据的依赖做比较。我们的目的不是为了证明软件工程忽视了证据（虽然基于有争议观点和迷信的软件工程无疑仍然存在），而是为了说明我们的学科缺乏一个组织、文化和技术基础，以支持对不同渠道知识的聚集和整合。Barbara Kitchenham在第3章中介绍了这项运动。

我们还可以看到其他一些对软件工程的思考和反省，用来改善软件工程实践的例子。比如Fred Brooks和Walter Vincenti这样的作家，通过提取经验并使用从实践和经验中来的例证来关注实践，以此思考工程活动的本质。我们不会仅仅因为他们的论文基于的不是科学实验而是反思经验，就抵制他们的重要性。但是我们会仔细思考他们的结论：我们是否同意这些结论，这些结论是否适合我们的环境，是否与我们的经验匹配。也就是说，它们是否与我们熟悉的证据相一致。

但不论我们依靠的是科学研究（剩余章节都会关注于此），还是开明人士的报告，我们认为，作为一个软件工程师，需要对证据有十分精确的理解，才能帮助我们做出正确的决定，才能根据我们想问的问题来评估对证据的需求。我们本质上并不需要更多的指标，或是广泛使用的科学方法，也不需要死板的方法论和教科书似的说教。强加的流程和方法本身不能产出质量，当然也不能保证质量。然而，我们通常试图用这些方法来达到真正的目标：系统性的、博识的、循证的批判性思维。

这个讨论的实质是让大家对证据有更多的思考：我们如何根据可信度和对目的的适用度来评估证据。

2.2　可信度和适用性

我们的讨论围绕着两个主要概念展开：可信度和适用性。

- 可信度

  你愿意（或应该）从多大程度上相信所提供的证据及其相关断言。可信度包括有效性，即研究及其断言在多大程度上能准确描述所关心的现象。有效性有如下几个方面。

❑ 你所观察到的是不是你想要观察到的，是不是你觉得正在观察到的；
❑ 所使用的度量是否真正度量了希望度量的东西；
❑ 是否准确解释了发生事件的原因；
❑ 我们是否能把已经研究过的东西推广到概念上可比较的其他环境中。

可信度需要一项研究来体现高度有效性，不仅如此，也需要良好的报告让读者知道在什么时候如何应用这项研究。

- 适用性

  在多大程度上你对证据及其断言感兴趣（或应该感兴趣）。在大多数情况下，你看都不会看那些不相关的研究。但有种典型的例外：你对问题感兴趣，但是找到答案的环境却和你的大相径庭。在那种情况下，适用性指的是在多大程度上你可以把研究结果推广到你自己的环境中。不幸的是，这通常是个很难回答的问题。有人把适用性看做可信度的一个方面，但我们相信把它们分开看待会有所帮助。因为低可信度的陈述相当于噪声（假设你还想要了解无数东西），而高可信度低适用性的陈述是高质量的信息，只不过你目前不觉得对你非常重要。但情况是会变的。

2.2.1 适用性，为什么使你信服的证据不能使我信服

证据不是证明。总的说来，任何足以让我们认为一个理由可信，或者比另一个理由更可信的实证就是证据。不同的目的需要不同标准的证据。有些目的需要强有力的证据。比如说，决定是否要把所有的软件开发都转换成面向方面的设计和实现，这需要非常有说服力的证据来证明收益大于投资（包括经济和文化方面）。有些目的只需要薄弱的证据即可。比如说，一个面向方面的追踪应用实例可能已经足够证明面向方面的编程方法（AOP）是有价值的，留待进一步研究来澄清的是它能（在什么环境下）多好地完成目标。一份以找出设计中的缺陷为目标的评估研究，可能只需要少数参与者的回复。我们知道有一个为制造业所设计的语音反馈系统原型，在初始用户指出他们用耳塞就能屏蔽工厂噪声之后就被放弃了。一些目的只需要反例就够了。比如，当一个人试图辩驳一项假设或者一项普适的断言时，他只需要一个反例。例如，可视化编程的支持者们声称“可以图形化的就是好的”，但是通过在一项实验中证明嵌套结构在用文本表示时理解起来更快，使得这些支持者们开始重新审视他们的看法。

所以，你想问的问题和特定环境下需要的证据之间是有联系的。你需要的证据和能提供那些证据的方法之间也是有联系的。比如说，你不能向提供信息的人询问不能说的秘密，这就是调查的局限性之一。软件是创建并部署在社会技术背景（socio-technical context）之下的，所以关于软件的断言和论点需要同时借鉴考虑了社会和技术背景的证据以及包含两者之间关系的证据。

例如，理解面向方面的设计对你项目的沟通结构所产生的影响，需要先观察那项技术的整个

应用场景。对方法的良好应用意味着“对症下药”，也意味着合理使用那些和问题相关并且服务于目的的证据。

2.2.2 定性证据对战定量证据：错误的二分法

研究中的讨论通常把定性研究和定量研究区分开来。概括来说，区别存在于所关注的问题中。定量研究围绕度量展开，通常见到的有比较问题（“A比B快吗？”）、假设问题（“如果A改变了，B会改变吗？”）和多少问题（“开发人员花了多少时间调试？”）。相反，定性研究围绕描述和分类展开，通常会问为什么（“A为什么比B学起来容易？”）或者怎样（“开发人员用什么方法怎样处理调试？”）。

如大部分模型一样，这只是个简化。一些关于概念上的区分会导致混乱，需要一些简短的讨论来澄清。

- 如果说定量证据通常比定性证据要好，那是无稽之谈（反之亦如此）。

  两种证据有不同的目的，因此不能直接比较。定性证据用来确定所关心的现象并解开已知所关注的费解现象。定量证据是为已确定的现象准备的，这些现象要么已经被理解得足够好，要么足够简单而能被独立研究。

  因此，定性研究应该在定量研究之前开展，查看那些更复杂的情况。当只涉及少数不同因素时（如物理学），就可以快速推进至定量研究了；当涉及许多不同因素时（如人类社会互动），这种转换要么需要更长的时间，要么需要利用尚不成熟的简化。软件工程证据中许多可信度的问题都来自于这种不成熟的简化。

- 这不是二分法，这是连续体。

  定性和定量研究不像简化中的那样独立。定性研究可能需要收集定性的数据（如言论或行动的记录），然后把数据进行系统化编码（如数据分类），最终量化编码数据（就是计算每个分类中的实例）并进行统计分析。定量研究反过来也可能有定性的因素存在。例如，当比较两种方法A和B的功效时，可能会比较两种方法的产品，把它们评估为“好”、“不错”和“差”。这些回答是定性的，它们落在顺序量表上。然而，可以继续使用统计方法（如Wilcoxon秩和检验）来确定A的输出是否显著地比B的输出好。研究结构是实验性的，而分析是定量的，和度量单位是厘米或秒用的是同样的技巧。（即使在使用性能指标的研究中，对指标的选择以及指标和质量的联系也可能是基于定性评估的。）

  软件系统中最有力的研究结合了定量证据（如性能度量）和定性证据（如流程描述）来记录现象，识别关键因素，并为现象和因素之间的关系提供有根据的解释。

- 定性研究不一定是“软的”，定量研究也不一定是“硬的”。

  优质研究的结果绝不会是武断的。好的科学寻求可重现的（“硬的”）结果，因为可重现的结果意味着它是可靠的，而且重现的过程使研究方法接受严格的审查。重现结果意味着重复（“复制”）各自的研究，要么在几乎完全相同的情况下（紧密复制），要么在不同但是想法类似的方式下（宽松复制）。定性研究几乎总是涉及独特的人文背景，因此很难紧密复制。

然而，那不意味着研究结果不能重现；这通常是可能的。从相关性的角度来说，宽松复制比紧密复制更有价值，因为这意味着结果更具普及性。从另一个角度来说，定量研究的结果有时不能重现。比如，John Daly对继承深度的研究被三个不同的研究组复制，得出了相互矛盾的结果[8]。

总结而言，定量研究的优势是用几句简单的陈述就能描述情况，因此，有时能把事情弄得很清楚。它们的劣势是忽视了太多的信息，以至于通常很难确定结果真正意味着什么以及何时适用。定性研究的长处是它们的结果反映并展示了真实世界的复杂度。劣势是它们因此更难评估。如果不清楚如何把研究结果映射到真实世界的环境下，任何研究都难以应用于真实生活，或是因为实验范围太过狭窄，无法普及，或是因为观测的背景太过不同。

“底线”是方法必须适用于问题。关于社会背景、流程的问题，以及人们相信他们所做的事和他们真实做过的事之间的差异，需要不同形式的调查，这不是算法性能能解决的问题。

2.3 整合证据

解决证据局限性的一种方法是结合不同形式或不同渠道的证据，就是把针对同样问题的不同研究结果整合起来。梳理证据背后的原因是，如果不同形式的证据或者不同渠道的证据能够呼应的话（或者至少不相互矛盾），它们就聚集了一种“重要性”，加在一起的可信度更高。

软件工程还没有证据的合并体。然而，已经有很多人在努力合并证据。Barbara Kitchenham及其同事们一直在支持系统性文献评审的运动，使用有明确标准的评估框架来检验并集合关于特定话题的已发表的证据。比如，Jørgensen和Shepperd在成本模型上的研究汇集并整合了比较模型和人脑之间估算性能的证据，并指出它们大体上是可以比较的[5]。有趣的是，系统性评审（以及Kitchenham等人实施的对系统性评审的第三方评审）暴露了证据基础中的引人注意的弱点[7]。

- 软件工程中的大部分话题几乎没有可信的证据。
- 对发布证据质量的担忧使努力得不到回报。
- 对汇报证据质量的担忧（如，是否充分并准确地描述了方法）限制了证据的评估。

然而，系统性评审不是验证研究结果的最终判定。它们的弱点之一是这种集合研究的方式使之很难对研究背景有适当的关注，而这在验证和应用研究时的重要性是被普遍公认的。可能产生的后果是，系统性评审很难处理定性研究，也因此经常把它们排除在评审之外，从而也排除了它们所提供的证据。另一个后果是，把不同背景的研究结果放在一起时（如，学生的实践和专业人员的实践），如果把它们的背景看成等同的话，就会有过度普遍化的危险。

方法论，包含方法的标准应用的统一调查系统，提供了让研究员比较和对比结果的有利条件，因此，证据能够随着时间而积累，稳定可靠的证据能为知识提供有力的基础。像化学这样的学科，特别是有详细说明的关注点和标准的问题形式的子学科，有定义完备的方法论。它们也可能有标准的汇报实践，通过标准的汇报形式来强制标准的方法论。

任何方法论都是一种镜头，研究者可以通过不同的镜头观察整个世界。但重要的是，必须承认不是所有的事物都在聚焦范围之内。对方法论的盲目追随会导致各种难堪的失误，特别是当研

究者不理解方法论背后的惯例和假设时。即使存在普遍接受的方法论，研究者也不能解除责任，放弃从所需证据的角度来证明技术选择的正当性。

基准测量是软件工程方法中基于集合证据产出适当结果的例子。它需要根据定义非常准确的程序（或基准）来度量性能。SPEC CPU基准是一个很好的例子，不管它的名字如何，它度量了CPU、内存子系统、操作系统和高CPU使用率应用程序编译器的组合性能。它包含了一系列应用程序源码，外加如何编译它们的详细指示，以及如何运行输入并度量它们的指示。

如果基准有明确的规定和适当的应用，你就能很准确地知道基准的结果意味着什么，也就不需要慌乱地比较Sun、HP、Intel之间的结果差异。这种可靠性和可比性正是发明基准的目的，也使它成为了可信度的标志。

那基准是免费的午餐吗？当然不是！对基准证据的真正顾虑是相关性：所使用的度量是否能代表所关注的现象并适用与此。基准所包含的内容始终是一个值得关注的问题，而且常常争吵不休。SPEC CPU从这方面来说是相当成功的，但论及其他，如关系型数据库管理系统（RDBMSes）事物处理负载量的基准TPC-C，就引起了很多的怀疑。

局限性和偏见

即使是高质量的证据也通常是片面的。我们常常不能直接评估一个现象，所以我们只能研究那些我们能直接研究的结果，或者只观注现象的一部分，或者从特定的角度观注它，又或者我们只观注那些我们能度量的东西，并希望它能映射到我们真正关心的东西上。度量是一种速记，是对现象的简洁表达或反映。但它通常不是现象本身；度量是一种有代表性的简化。高可信度需要证明所做选择的合理性。

更糟糕的是，证据是会有偏见的。多少软件工程师会相信那些老套的清洁剂广告中的消费者实验和“盲测”（“Duz能清洁更多碗盘……”）？广告法规要求这种消费者实验必须遵循一定的标准使条件具备可比性：同样的污垢、同样的用水量、等量的清洁剂等。但是广告商可以任意制定条件。他们可以优化一些条件，如污垢的种类和适于产品的水温。“哈！”我们会说，“偏见是与生俱来的。”而许多发表的软件工程方法和工具评估仍然遵循了同样的模式：无论是有意的还是无意的，评估的背景是有所设计的，以证明所推销的方法或工具的优点，而不是基于独立定义的有根据的标准，公平地与其他工具和方法做比较。

当证据悄悄地被玷污和妥协时，偏见就产生了。这是因为之前没有考虑到的因素导致了结果扭曲，这样的因素包括如其他影响、合并变量、不适当的度量、或者对样本的选择不具代表性。偏见会对研究的有效性产生威胁，所以当我们在评估可信度的时候会寻找可能存在的偏见。

我们不只需要理解特定证据的价值和局限性，也要了解不同形式的证据如何比较，以及他们如何能相互组合来补偿各自的局限性。

2.4 证据的类型以及它们的优缺点

让我们来用一个例子来更好地说明。想象我们正在评估一种新的软件工程技术“AWE”（A

Wonderful Excitement，美妙的新方法），它被用来替代“BURP”（Boring but Usually Respected Predecessor，枯燥但被口碑良好的旧方法）。我们需要考虑哪种证据来决定是否采用AWE？在接下来的章节中，我们会描述常见类型的研究，并根据可信度和适用度的典型问题对每个类型进行评估。

2.4.1　对照实验和准实验

对照实验在我们需要对两种或多种条件进行直接比较时适用（如使用AWE相对于使用BURP），它基于一个或多个有可靠度量的标准，比如完成特定任务所需要的时间。这些实验在度量很棘手时也很有帮助，比如对AWE和BURP所产出的产品中的缺陷计数。“对照”意味着保持所有（除了把AWE换成BURP以外）别的东西为常量，这点我们可以直接根据工作环境和所要解决的任务来做到。对于所涉及的庞大的人为因素变量，实施对照的唯一方法是使用一组对象（而不是一个对象），指望所有的差异会在组内平均化。如果我们能把对象随机分组的话（随机实验），这种希望是被证明合理的（至少在统计意义上）。当然只有所有对象使用AWE和BURP的能力相当时，这才有意义。随机实验是能证明因果关系的唯一研究方法：如果我们只变动了AWE和BURP，那结果中的任何改变（除了统计波动）都是由这个差异所产生的。

有时候，我们不能随机分配开发人员，因为我们只能对已经存在的团体进行研究。比如，假设我们正在比较C、C++、Java、Perl、Python、Rexx和TCL语言（如第14章所述）。你知道多少对所有7种语言都同样精通的程序员？对一项随机化实验来说，你需要很多这样的人来填满7组！培训足够的人到达那个水平是不实际的。在这种情况下，我们使用现有的团体（准实验），但需要必须考虑的是，比如是否所有的聪明才子都只用B或P开头的语言。这种情况意味着我们的团体反映了不同能力的人群，因此会歪曲比较的结果。（虽然，我们也可以把它归因为语言本身的因素。）

1. 可信度

给定了一个清晰良好的假设、干净的设计和实现、以及合理的度量，对照实验的可信度通常很高。因为理论上来说，环境设置是公平的，对结果的解释也很清晰。实验中典型的可信度问题包含以下几点。

- 对象偏差

  研究对象是否对AWE和BURP的能力相当？如果不是，实验可能更多地反映了参与者的特征而不是技术的特征。把这个问题稍微变化一下，就会微妙地涉及动机问题：BURP很枯燥而AWE很让人兴奋，但长期来看就未必如此。如果实验者恰巧是AWE的发明者而因此对AWE非常热衷的话，差异就会清晰地显现出来。

- 任务差异

  被这样的实验者选出的任务往往会突出AWE的优势而不是BURP的优势。

- 小组任务分配偏差

  小组的既有差异造成对结果的歪曲经常是准实验中的一项风险。

2. 适用度

这是实验真正的弱点。好的实验设计是精确的，限定的实验环境会使结果难以运用至外面的世界。实验容易出现“沙子漏过手指”的现象：一个人可能抓住了一个概念，如生产力，但当概念通过许多优化后，能在有足够控制的环境中成功映射于特定的度量时，它似乎已近溜走了。

2.4.2 问卷调查

问卷调查：就是向许多人询问关于一个事物的同样问题。它是度量态度的可选方法。我认为AWE好在哪里？我个人发现BURP乏味在哪里？为什么？社会学和心理学已经制作出了一个详细的（而且昂贵的）方法论，来找出能正确度量针对特定问题态度的合适问题。不幸的是，这种方法论在软件工程调查中通常被忽视了。

调查也可以用来收集经验（可靠性稍低）。比如，AWE或者BURP中最易出错的五件事。调查可以很好的规模化，是涉及大量对象的最便宜方法。如果它们包含开放性问题的话，可以作为定性甚至定量研究的基础。

1. 可信度

虽然调查既便宜又方便，但很难通过它们达到高可信度。典型的可信度问题包括：

- 不可靠的问题

  模糊、有误导或模棱两可的问题会在不同回答者的心中产生不同的解读，导致无意义的结果。这些问题可能很细微。

- 无效的问题或结论

  这个问题特别会出现在误用调查来获取复杂的实际问题时，“你的AWE图通常包含多少缺陷？”人们不能给出准确的答案，因为他们的了解不足以回答这个问题。如果我们就事论事的话，这个问题不会太大：答案揭示了回答者的想法，正如我们之前强调的那样，它反映了他们的态度。然而更常见的是，答案被看成真正的事实，那相应的结论就会出错。（在这种情况下，评论家会经常把调查评论为“太主观”，但是调查就应该是主观的！如果主观是一个问题的话，这个方法就是被误用了。）

- 对象偏差

  有时，回答者不会回答他们真实的想法，而是因为社会政治的原因压抑或夸大了某些问题。

- 不明确的目标群体

  调查的结果可以推广至任何被调查者的样本所能代表的目标群体。这个群体总是存在的，但是研究者很少能成功地描述它。

概括说来，可信度要求两个条件。第一，调查必须发给一个定义清晰的、易于理解的群体。“从日期D到日期E中，所有网上论坛X、Y、Z的读者”是清晰的定义，但不易理解；我们不清楚这群人到底是谁。第二，群体中必须有大部分人（50%以上为佳）回答了问题，不然，回答者就可能是有偏差的样本。也许群体中4%的AWE粉丝（“唯一让我在工作中保持清醒的东西”）和5%的BURP厌恶者（“太讨厌了以至于我犯了两倍的错误”）最有动机参与，但是能让他们代表一半

的样本量吗？

2. 适用度

调查能解决的问题通常不是我们最关心的问题。回答形式的限制通常不能令人满意，也很难被满意地解读，所以通常也很难有更普遍的运用。然而，调查可以很有效，它可以提供补充信息并为其他形式的证据提供背景信息：如提供实验参与者的背景信息，或者提供满意度和偏好信息来补充性能数据。

2.4.3 经验汇报和案例研究

既然实验和调查有局限性，而且我们对基于现实世界经验的评估感兴趣，也许仔细深入地观察一到两个实施案例会提供我们做决定所需要的信息，或者至少把我们的注意力聚焦在我们先需要回答的问题上。

案例研究（或者它不那么正式的亲戚经验汇报）描述了一个现象（通常是一系列事件及其结果）在真实软件工程环境下的特定实例。原则上来说，案例研究是运用详细而精确的方法论的结果，但是这个术语通常被更宽松地使用。虽然现象本身是独特的，描述它时，我们会希望其他情况和它足够的接近，那它才会有意义。

案例研究利用许多不同类型的数据，如研究者通过不同方法（如直接观察、面谈、文档分析以及特殊目的的数据分析程序）收集的谈话、活动、文档和数据记录。案例研究可以针对大范围的问题，如AWE流程中最大的问题是什么？AWE如何影响设计活动？它如何影响设计结果？它如何组织测试流程？等等。

在我们的AWE对比BURP的例子中，BURP可以单独作为一个案例来研究，也可以作为同一个研究的第二个案例，研究者尽其所能地匹配调查和讨论的结构。如果描述地足够具体的话，我们就有可能把这些观察结果运用到我们的环境中。

1. 可信度

对于经验汇报和案例研究来说，充满了可信度的问题，因为这些研究是特定于它们的背景的。大部分的情况都与以下有关：

- 选择性的记录；
- 含糊不清的表达。

因此，准确描述设置和数据并小心解读研究结论是最重要的事情。案例研究的可信度关键在于汇报的质量。

2. 适用性

通常，研究者和出版者发表经验汇报和案例研究，是因为相信它们的普遍适用性。解读适用性很具有挑战，因为不同的设置之间会有许多重要的不同点。但这种汇报中所提出的问题的适用性是公认的，即使结论的相关性难以确定。

2.4.4 其他方法

之前的章节并不是无一不包的。比如，我们也许会使用基准方法评估支持AWE和BURP延展

性的工具，或者用用户研究来评估易学性和易出错性。这些方法（和其他方法）都有自己的可信度和适用性的问题。

2.4.5　报告中的可信度（或缺乏可信度）的标识

2

如果了解了之前章节所讨论的基本原则问题之后，你会发现可信度很大程度上可以归结为做得好或做得不好的很多方面。现在我们将讨论在如何从这些方面来分析研究报告的可信度。

1. 一般特性

对于一个高可信度的研究来说：

- 它的汇报是详细而准确的（而不是含糊不清的）；
- 讨论的形式是诚实的（而不是搞搞门面功夫），逻辑和论点都是正确的（而不是不一致的或是假的、不合逻辑的）；
- 研究的设置能最大化适用性（而不是过于简单或过于专门化）；
- 虽然本身很枯燥，但是好的报告的行文应该尽量生动（而不是很无趣）。

2. 清晰的研究问题

表述清晰明确的研究问题是可信度的基础，因为如果作者不解释他们在寻找什么，那他们还能拿出什么东西让你相信呢？研究问题不需要大张旗鼓地宣告，但必须可以从概要和简介中清楚地辨别出来。

通常，研究问题是可以被清晰地辨别出来的，但是问题本身却很含糊。在这种情况下，结果要不就是也很含糊，或者会带有意外性和随机性。这类研究也许读起来很有趣，但可信度达到中等就不错了。

3. 对研究安排的信息说明

理解研究的设置是可信度的核心，这也是新闻稿和科技报告的主要区别。即使是最吸引人的结果，如果它是通过盯着水晶球看出来的，那也不会有可信度。以下的研究结果一定很吸引人：“600人参与的实证研究显示Java在各个方面都比C++好：编程时间缩短了11%，调试时间缩短了47%，长期设计稳定度提高了42%。只有在运行性能上，C++仍高出Java 23%。”

但如果你知道这些结果是通过问卷调查而得出的话，那可信度就大打折扣了。如果你仔细查看问题的话，可信度会继续降低：他们是如何在程序完全不同的情况下比较编程和调试时间的？哦，他们问了任务完成的时间比预期时间长的频率！那“长期设计稳定度”是什么？哦，他们问了方法中有多少部分是从来不变的！问题都出在细节上：方法、样本、数据、分析。

粗略的凭经验来说，你可以完全忽略那些没有描述设置的研究，对那些设置的描述使你产生好奇疑问的研究，你也要保持怀疑的态度：这是哪个类型的研究？研究对象着手于哪些任务？在哪种工作环境下？研究对象是谁？数据是如何收集的？数据是如何验证的？主要的度量定义到何种精度？一份优秀的实证研究报告能令人满意地回答所有这些问题。

4. 有意义并能让人理解的数据呈现

当你知道研究是如何构造的、数据是如何收集的时候，你需要进一步了解关于数据本身的信息。研究报告可能没有空间来发布原始数据，所以，即使是很小的研究也会通过统计学家所说的

描述统计学来总结数据。

假设有一项涉及6个不同小组和5个不同度量的研究。一些作者会用表格来呈现他们的数据，每行列出小组名字、小组大小，还可以列出针对一个度量的最大值、最小值、平均值、标准差等。表格被分成5块，每块包含6行这样的数据，总共有37行，占据半页以上的空间。当你看到这样的信息时会怎么做？你可以忽略这个表格（因此就不能细查数据），或者你可以试图针对每个度量仔细比较不同组的数据。在后者的情况中，你会在脑子里创造一副全景图，首先得想明白个体条目，然后形象化它们之间的关系。用更有趣的话说，你会掘出数据的坟墓，让它们复活，然后按照你的音乐跳舞。

我们更希望作者们保持他们数据的原样，把它们通过某种方法表达出来，帮助读者的介入：通过找到与作者想要讨论的关键问题和关系相关的直观表示，以及与表格和文本数据显示直接、清晰相关的可视化呈现。良好的数据呈现包含突出视觉效果（如，标尺、色彩），能前后一致地表达出所关注的数据。差得数据呈现所使用的可视化品质会使观众对数据的理解产生转移或模糊，比如，通过用线段连接离散数据来表达某种连贯性，或者通过在相关图表上使用不同的比例尺或者使用起始点不为0的标尺，或者通过添加比原始数据更突出的外来的元素。Edward R. Tufte有一本关于如何正确进行数据可视化工作的开创性著作[9]。

5. 透明的统计分析（如果有的话）

统计分析（演绎统计学）说的是从“信号”中分离“噪声”：把可能是从巧合或错误中得出的结果分离于从有意义的研究现象中得出的结果。运用统计的目的是降低解读结果的不确定性。但是，许多作者把统计分析作为一种威胁。他们列出所有的Alpha级别、p值、自由度、剩余平方和、M参数、Σ、θ、β系数、ρ、τ等所有一切。这只是为了告诉你：“如果你敢质疑我的观点，我就会用我的显著性测试来砸你的脑袋。”可信的研究使用统计数据来解释和确保结果，差的研究使用它们来混淆视听（因为作者需要隐藏弱点）或恫吓他人（因为作者自己不能确定这些统计戏法的意义）。

在好的研究中，作者会用简单的语言解释他们所使用的每个统计推论。他们更喜欢使用易于理解的推论（如置信区间）而不是难以解释的推论（如p值和效能，用标准差归一的效应量）。他们会清晰地用如下语句解读每个结果：“这里也许有一些真正的差异”（正面结果），“这里似乎没有影响，或者只有很少的影响；我们看到的大部分是随机噪声”（负面结果），或者“还不清楚其意义”（空结果）。即使是最后的那种结果，也很令人安心，因为它告诉你在看到数据时对其意义的不确定是有原因的，即使是统计推论也不能排除这个不确定性（至少不能通过这个来排除；也许有不同的分析可以带来指路明灯）。

6. 诚实地讨论局限性

任何实证研究的坚实汇报都需要有一个独立章节来讨论研究的局限性，通常以“对有效性的威胁”为标题。这个讨论提供关于以下问题的信息：“什么是通过这个研究所不能达到的”，“什么样的解读会有问题（构建效度）”，“研究中的什么东西可能或者已经出错了（内部效度）”，“要推广研究结果的限制是什么（外部效度）”。对一份好的研究报告来说，你通常已经意识到了这些问题，那这个章节就不会提供许多令人惊讶的信息。可信研究的作者能接受批判点的存在。

如果一份研究试图消除所有批评的可能性，通常不是一个好的兆头。

如果做得好的话，最有趣的部分通常是对可推广性的讨论，因为这直接和结果的适用性相关。好的报告会同时在不同方面针对不同的推广目标领域提供正面和反面的可推广性观点。

7. 坚实且适用的结论

实证研究报告的挑剔读者们会基于他们自己对所呈证据的评估形成自己的观点，不会单纯依赖于作者在摘要和结论中提供的陈述。

如果作者的结论过分吹嘘了结果（把结果推广至没有坚固论点支撑的领域，或有甚者，基于只与研究现象有粗略联系的现象得出结论），你就更应该对报告其余部分的可信度提高警惕。你尤其应该拿起一支最粗的红笔，划掉摘要和结论，那你就绝不会在重新整理你对特定研究的记忆时依赖它们了。请记住很少有读者这样做，而所有人都需要重新整理他们的记忆。因此，很多在报告或课本中引用的研究会错误地指向夸张的结论，就像它们是真的一样。（人无完人，科学家也不例外。）

如果是另一种情况，结论看起来很有思想，明显试图在可信度和适用性之前寻找一个平衡的话，它们的可信度会加固，适用度会被最大化。这样的作者把研究的结果和限制同时放在一个秤盘上，而把所有（从文献中其他结果的角度来看）的普遍化问题放在另一个秤盘，然后告诉你他们所想的可能是正确的归纳。这样的信息是有价值的，因为作者有很多在文章中没有明显出现的研究信息，但却在判断的时候使用到了。

2.5　社会、文化、软件工程和你

目前为止，你大概会同意达到高可信度绝非易事。然而，这并不意味着没有（或几乎没有）可靠的研究；只是可靠的研究总是很少。它们比我们想象中的专业得多，而且充满了我们不喜欢的“如果”、“当”和众多假设。在这样的情况下抱怨是没有意义的；这只是我们所生存和创造（就技术而言）的复杂世界所造成的不可规避的结果。如果我们足够耐心，并且能对我们已经发现的东西感到高兴的话，那这就不是一个问题。

我们认为，真正的问题在于：虽然工程师和科学家们很理解复杂性，也很重视复杂性及其带来的工作量，而且还能对它表示敬畏，但是我们整体的社会和文化并不是这样。我们被许多壮观的事物和景象所包围，所以不再把小新闻当做新闻。我们很难去留意那些由50个单词组成的未经扭曲的、错综复杂的研究结论。

大众媒体会因此有所行动。为了吸引眼球，他们忽视、夸张或扭曲实证研究的结论，这些结论通常被弄得面目全非。科学家们通常也帮不了什么忙，只能写写所谓的摘要，仅仅是宣布结果而不是总结概括它们。在任何情况下，担子都会压在挑剔的读者肩上，需要他们更仔细地阅读。你需要从研究中挖掘出一份报告，消化它，决定它的可信度，把对你可靠和相关的东西带回家。作为软件工程师你的资质意味着你有能力这样做。软件工程的进步需要许多工程师经常实践这种能力。这本书的内容就代表着一次实践的大好机会。

观测研究已经证明的一件事是：软件工程专家会收集与之相关的证据，根据它们的目的确定

所需的可信程度。也就是说，如果它们在为专业音响设计数字过滤器，它们可能会做一个基于输出的对照实验，来确定软件是否产出了人耳所需的音质。如果他们正在和市场部争论使用模拟控制的客户是否有处理虚拟化接口的准备，他们可能会把一位研究员派去现场讨论“什么是客户所控制的参数”，“他们的现有系统如何运作”，“客户认为他们的任务如何”等。如果他们正在设计安全性至关重要的前端系统，他们可能会做一次客户研究来确定几个设计选择中的哪一个能最好地适应人机交互。如果他们在设计安全性至关重要的后端系统，他们可能使用正式的方法来建立对特定需求的正确实现。如果他们在优化排程演算法，他们也许会对工业投入运行基准。

所以我们不该拒绝各种形式的证据，我们应该拥抱它们，努力思考并保持批判性。

2.6　致谢

本文的作者们感谢他们的同事David Budgen、Gordon Rugg和Judith Segal，他们容忍了作者在为本文采证过程中的犹豫不决。

2.7　参考文献

[1] [Brooks 1975] Brooks, F. 1975. *The Mythical Man-Month.* Boston: Addison-Wesley.

[2] [Eisenstadt 1993] Eisenstadt, M. 1993. Tales of Debugging from the Front Lines. In *Empirical Studies of Programmers: Fifth Workshop*, ed. C.R. Cook, J.C. Scholtz, and J.C. Spohrer, 86-112. Norwood, NJ: Ablex Publishing.

[3] [Finkelstein 2003] Finkelstein, A. 2003. A call for evidence-based software engineering. Keynote address to Psychology of Programming Interest Group Annual Workshop, Keele University, April 8-10, in Keele, UK.

[4] [Green and Petre 1992] Green, T.R.G., and M. Petre. 1992. When visual programs are harder to read than textual programs. *Proceedings of the Sixth European Conference on Cognitive Ergonomics*: 167-180.

[5] [Jørgensen and Shepperd 2007] Jørgensen, M., and M. Shepperd. 2007. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Transactions on Software Engineering* 33(1): 33-53.

[6] [Kitchenham et al. 2004] Kitchenham, B.A., T. Dybå, and M. Jørgensen. 2004. Evidence-based software engineering. *Proceedings of the 26th International Conference on Software Engineering (ICSE)*: 273-281.

[7] [Kitchenham et al. 2009] Kitchenham, B., O.P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. 2009. Systematic literature reviews in software engineering—A systematic literature review. *Information and Software Technology* 51(1): 7-15.

[8] [Prechelt et al. 2003] Prechelt, L., B. Unger, M. Philippsen, and W.F. Tichy. A Controlled Experiment on Inheritance Depth as a Cost Factor for Maintenance. *Journal of Systems and Software* 65(2): 115-126.

[9] [Tufte 1983] Tufte, E.R. 1983. *The Visual Display of Quantitative Information.* Cheshire, CT: Graphics Press.

[10] [Vincenti 1993] Vincenti, W.A. 1993. *What Engineers Know and How They Know it: Analytical Studies from Aeronautical History* Baltimore: Johns Hopkins University Press.

第3章

我们能从系统性评审中学到什么

Barbara Kitchenham

我在强烈提倡循证软件工程[16] [30]的同时，也强烈提倡*系统性评审*（SR）[26] [28]。这在软件工程中有时被称为*系统性文献评审*，以避免与检查方法混淆（如，阅读和审查软件工程文档和代码的方法）。如果没有健全的方法论来集合不同实证研究的证据，那循证软件工程就不可能存在。系统性评审恰恰提供了这种方法论。

系统性评审已经在其他学科中被广泛应用了几十年。当研究人员需要调查针对一个特定“话题”的所有支持性或反驳性证据时，就需要启动一次系统性评审。在软件工程中，通常会涉及打听一种方法或者流程的效果。实施系统性评审的研究人员会选择与特定研究问题相关的实证研究，评估每个研究的有效性，然后确定这些研究所显示的趋势。因此，系统性评审旨在以一种公平、可复验、可审查的方式，找到、评估并集合所有关于某个话题的相关证据。

这一章会带着对实证软件工程的普遍关注来向读者介绍系统性评审的价值。我也力求帮助见习研究人员（如博士生）开始使用系统性评审，他们可能正在寻找可靠的方法来着手能代表领域发展水平的评审。这一章应该也会对更有经验的实证研究者有所帮助，他们也许对系统性评审的方法论的价值还没有足够的信心。许多欧洲研究者已经开始发表软件工程的系统性评审，但来自美国的研究者相对较少[33] [34]。

我也希望这一章内容能提醒实证研究者，他们的研究可能也会为将来系统性评审做出贡献，并因此能在汇报他们结果的同时谨记将来研究证据的集合。最近的一次系统性评审发现，很难开始一个完整的元分析，因为个体的原始研究在汇报他们的结果时使用了完全不同的实践方法[51] [52]。

在循证软件工程的背景下，系统性评审的目标不只是为研究者提供方法论，而是影响实践。因此，我希望业界的管理者和决策者也能在这章中找到一些与他们需求相关的东西。业界最大的教训是，“常识”和专家观点不应该是决定选择软件工程方法的唯一基础。此外，很不幸的是，不能默认地信赖个体的实证研究。面对关于采用新方法的重要决定时，决策者需要不带偏见地对所有的相关证据的概括。系统性评审恰恰提供了交付这种概括的方法。

3.1 系统性评审总览

可信的调查始于*原始研究*：带有与所研究问题相关的定性和定量结果的实验。一次系统性评审集合了不同独立实验的结果，有时会运用统计学的*元分析*。

证明系统性评审必要性的典型例子来自医学领域。在1990年，Crowley等人发布了一份关于对将要早产的孕妇使用皮质激素效果的系统性评审，其中包括一份对于12项原始研究的元分析[11]。皮质激素被认为能减少早产儿的肺部问题。Crowley等人的系统性评审确认了皮质激素的使用大幅度地减小了新生儿的死亡风险。在那个时候，皮质类激素并不是治疗早产儿的标准方法，系统性评审的发布改变了医疗实践。

然而，这并没有被认为是循证医学的重大胜利。在那12篇研究中，8篇是在1982年之前发表的。如果那8篇研究能在1982年集合的话，8年的错误治疗和与之相关的新生儿死亡就能够被避免。这引发了对及时集合证据的重要性的重新评估，以及Cochrane协作组织的建立，该非营利组织旨在实施针对医疗和保健问题的系统性评审，并维护系统性评审报告数据库。

你可以在Cochrane协作组织的知识库中找到免费的Crowley等人对皮质类激素报告的更新[43]。这不只是为了记录历史；对那些新开始系统性评审的研究员来说，这也是一个很好的优质评审的例子。

你也许会问系统性评审有什么创新。毕竟，软件工程研究者已经持续多年产出不落后于时代的报告，并且在汇报他们最新研究成果的同时，也总结了相关的工作。要回答这个问题，我们得先看看其他使用系统性评审的学科（例如，心理学、社会学和医学），它们已经找出了由于缺乏正规方法论而导致的传统评审中的无数问题。

- 专家也会犯错

  Antman等人确认了专家观点不总能反映当前的医学知识状态[1]。

- 研究者所选择的“相关研究”可能带有偏见

  Shadish调研了超过280篇心理学期刊文章的作者。他发现文章经常引用某些研究的原因并不是因为它们的高质量，而是因为它们支持了作者的观点[45]。

- 非正规评审可能缺失了重要的研究

  比如，一份非正规的评审说服了诺贝尔奖得主Linus Pauling，大剂量的维生素C可以抵抗常规感冒。然而，一份系统性评审得出了与之相反的结论，并发现Pauling并没有引用15篇有可靠方法论基础的研究中的至少5篇[36]。

系统性评审的主要优势是，他基于定义完备的方法论。开展一项系统评审所涉及的最重要的流程包括如下几个方面。

- 明确叙述所研究的问题

  这是任何系统性评审的起始点。这需要把一个公认的信息需求，提炼至能被当前研究文献所回答的一个或多个问题。

- 找到相关的研究

  这需要一个研究团队尽可能全面地搜索研究文献，来找到解决所研究问题的文章。有时

候，需要单个研究人员（如一个博士生）独自完成这项任务，但是这一限制可能导致漏掉一些相关的文章。最佳实践是，研究过程中识别出的每篇文章都至少被两名研究人员检查过，他们各自评估这篇文章是否应该被系统性评审所包含。对单篇文章的不同意见必须在讨论后解决（如果需要的话，可以引入第三名团队成员）。

系统性评审的有效性依赖于找到足够的实证研究来解决所研究的问题，并且证明研究过程符合系统性评审所要求的完整性级别。

- 评估单个研究的质量

  这需要确定合适的标准来评估每项研究方法的严谨性，并依此确定它是否有资格被包含到文章的集合中。我们希望我们集合的文章是基于最高质量的证据的。质量评估是困难的，所以多数标准建议由至少两名研究者来评估每项研究的质量，并且通过讨论来解决不同意见。

- 提取并集合数据

  这需要提取每篇文章中针对所研究问题数据，然后适当地集合数据。有时，可以通过一次正式的元分析来集合数据，但在软件工程和其他非医学领域中，每篇研究的结果通常只是以表格的形式列出来，以确定潜在的趋势。与质量评估一样，数据提取通常由两名研究者来完成，他们讨论并解决所有的不同意见。

虽然我之前批评了对专家观点的依赖，但是我所见过的关于软件工程的最好的系统性评审，是由包含领域专家的团队执行的。领域专家的知识在系统性评审中可以被很好地加以利用，来确定需要搜索的专业会议和期刊，并确定可以用作基准的一组初始研究文章，来检查任何自动搜索过程的有效性。

对于刚开始学习系统性评审流程的新研究员，我推荐他们考虑*图谱研究*（mapping studies）。这类系统性评审试图寻找并分类涉及更宽泛研究话题的文献，而不是回答一个特定的研究问题[35]。这两类研究的区别可以通过比较Magne Jørgensen的两篇文章而看出。第一篇是传统的系统性评审，调查了成本估算模式的评估在预计项目成本时是否比专家判断的评估更精确[22]。这篇文章在3.3.1节中有所讨论。第二篇是高质量的图谱研究，对成本估算文献进行了分类[24]。

下面的3.3节讨论了一些挑战了软件工程 “常识”的系统性评审，证明了我们用健全方法论来集合证据的必要。然而，在讨论这些例子之前，我先总览一下系统性文献流程。

只对系统性评审的结果感兴趣，而对系统性评审的具体流程不感兴趣的读者，可以直接跳至3.3节。这章的简介应该足够你了解系统性评审流程的严谨性，不用你多花力气去读这么多的细节。

然而，如果你是一个新的或者有经验的研究者，希望评估系统性评审的价值，那你就需要阅读下一节。它讨论了系统性评审流程中的具体步骤，并告诉你这一方法论中所固有的一些实践困难。

3.2　系统性评审的长处和短处

随着时间的推移，我已经为软件工程的研究者们定义了两套执行系统性评审的标准。第一套

标准基于医学系统性评审[26]，而第二套在第一套的基础上进行了修订，把社会学研究中的新兴想法也纳入其中[28]。①

这一节简短地概述了这些报告所记录的指导方针。然而，我提醒读者不要以为他们可以基于这些简短概述就能成功地开展系统性评审。你应该阅读完整的技术报告，参考其他文献（比如，Fink[17]，Higgins和Green[20]，Khan等人[25]Petticrew和Roberts[41]的报告），阅读系统性评审的案例，包括这章晚些时候会讨论的软件工程的案例。如果你能查看一些好的系统性评审的例子，你就会明白它们不是简单形式的研究。它们需要花很长时间，也需要谨慎地计划、执行和汇报[7]。

3.2.1　系统性评审的流程

一次系统性评审有三个主要阶段：计划、执行评审和汇报结果。

1. 计划评审

这一阶段的步骤包括以下内容。

- 确定评审的必要性

  确定必要性能帮助确定评审的背景环境，能对所研究的问题给予约束，或者对包含或排除原始研究的标准给予限制。例如，如果你需要确定引入新技术的最佳方法，也许你会限制只包含产业领域的研究。此外，你应该检查是否有现成的文献已经对此话题领域进行了评审，如果有的话，需要证明另一次评审的必要性。

- 授权评审

  医疗和社会研究的政策制定者们通常会对当前公众感兴趣的主题授权系统性评审。一个基金组织通常会邀请了解主题本身也了解系统性评审的专家来提交实施评审的提议。然而，我从未听说“授权评审”这一步骤在软件工程中发生过。

- 指定所研究的问题

  对问题的选择会影响整个评审流程，所以需要有清楚的书面表达，并让所有的评审团的成员都理解。

- 建立评审协议

  这一步骤需要确定出开展评审和汇报结果时将要使用的所有流程：具体研究流程，包含和排除原始研究的标准，以及提取和集合的流程。这些都需要先经过检验来确保其适用于将来的任务。评审协议是用来减少研究偏见的一种方法（因为研究策略是提前定义的，不在任何一个研究员的控制范围之内），并能增加评审的可审计性和可重复性（因为计划的流程需要被充分报告）。这是一项非常耗时的工作。

- 评估评审协议

  多数标准建议团队之外的研究者来评审协议。然而，在实践中很难找到外部的研究者愿意担任这一角色，所以可能只会有一次内部的评审。

① 这些报告了其他经验性软件工程的资料可以在网站http://www.ebse.org.uk找到。

2. 执行评审

这一阶段的步骤包括如下内容。

- 找到相关的研究文献

　　这首先需要确定文献来源以搜索论文，然后确定如何组织搜索，最后确定相关研究论文。其中会涉及的主要问题包括：选择哪些数字图书库、索引系统、刊物和大会记录来进行搜索；搜索流程是自动化的、手动的、还是混合的；如何保证搜索流程的有效性。找到相关的研究是另一项耗时的活动，我们之后会详细讨论。

- 选择原始研究

　　选择的流程指，运用包含和排除的标准来决定搜索流程确定的每个研究是否被包括在评审中。

- 评估研究的质量

　　需要对原始研究进行质量评估。这样才能只包含有健全方法论的研究，或者当只有少量研究存在时，对读者给予提醒。通常你需要一份质量评价问卷，对实证研究的严谨性进行提问。

　　比如，以实验做例子，标准的问题包括："研究对象是否被随机分配于治疗中？""研究者是否在实验结束之前不知晓研究对象被分配于何种治疗中？"对于软件工程中的系统性评审，可以使用许多不同的研究方法。我推荐用Dybå 和Dingsøyr制订的清单[14]作为起始点，虽然我更倾向于使用顺序量表而不是二元选择来回答问题。你也应该参考第2章。

- 数据抽取和综合

　　从每篇文章中抽取出的数据必须在集合后才能回答所研究的问题。如果有足够的可比较的定量数据，可以使用元分析方法[37]。如果没有的话，结果应该制成表格，以某种方式应对所研究的问题。

　　元分析是医疗界的系统性评审中最常使用的数据集合形式，但在其他领域中却不那么常见。实践中，只有当每个原始研究的结果都以适当的统计分析呈现的时候，元分析才有可能性。然而，即使统计结果存在，也不是总能执行完整的元分析[52]。

3. 汇报评审

这一阶段的步骤包括如下内容。

- 指定分发机制

　　一些系统性评审非常冗长，需要考虑提交给期刊的文章是否需要有更详细的技术报告作为支撑。

- 主报告规格化

　　系统性评审遵循经验性论文的普通报告标准。我推荐使用结构式概要（如Roberts和Dalziel[43]报告中所用），因为有证据显示这是总结研究结果的有效方法[8]。如果你一直坚持使用一个好的模式，而且你的搜索和数据提取流程保持了良好记录，这是一个比较简单的任务。

- 评估报告

  标准流程推荐外部评审，但不管系统性评审是在内部还是外部被评估，评审者需要确认系统性评审的流程被完全写在报告中，而且所研究的问题和研究的流程、数据抽取流程、数据集合流程之间有直接的对应关系。

系统性评审要求一点：结果是可重复的，就是说如果另一个研究小组遵循同样的协议，他们会得到同样的结果。可重复性依赖于所研究的问题本身的清晰明确，不能模棱两可，也需要对搜索和集合的流程以及研究的范围进行完全的汇报。最近有个研究比较了两个针对同样研究问题的独立系统性评审[38]。这份研究总结：在原始研究数量较少而评审团由领域专家组成的情况下，软件工程的系统性评审可重复性相当高。另一份基于非常庞大的经验性文献库的系统性评审[51]发现了不同文献中的评审有巨大的差异，但它同时解释说这些差异来自于不同评审中隐含的研究问题之间的差异。

3.2.2　开展一项评审所牵连的问题

理论上说，系统性评审应该严格按照预先定义好的研究协议展开。然而，这并没有看起来那么容易。虽说协议是被测试过的，但是它可能没有识别出设计和汇报相关原始研究的所有变种。所以，你可能碰到协议所没有包含的情况。在这种情形下，协议需要修订。根据不同修订的性质，你可能需要评审以前的工作甚至重做许多工作，来保证其符合修订后的协议[51]。

在确定相关研究的过程中，关键因素是选择进行搜索的数字图书馆，还要确定搜索是自动的还是手动的。手动搜索需要查看一组（纸面或者在线）期刊的以往发行记录，然后从标题和摘要中确定哪些文章能够被候选包含在评审中。自动搜索使用字符串，通常是基于复杂布尔公式的字符串，通过在线目录查找文章。在医学文献中，建议从所研究的问题中抽取字符串做搜索。

对自动化搜索来说，Hannay等人推荐仅在ACM数字图书馆、IEEE Xplore、Compendex和ISI科技网上搜索资料，因为这些书库确保涵盖了ACM、IEEE、Kluwer Online、Science Direct、SpringerLink和Wiley的资料[19]。作者们也推荐对重要的主题大会刊物进行手动搜索。然而，如果你需要找到所有相关的文献，无论发表的还是未发表的，你也应该使用Google学术和CiteSeer。此外，不同的数字图书馆在搜索方式上有一些细微的差别。所以，你应该向图书管理员咨询你的搜索流程和搜索字符串，他会给你一些好的意见。相关的问题包括如下内容。

- 软件工程数字图书馆有不同的界面和对复杂布尔公式的不同容差，这在系统性评审的研究者寻找相关文章时会经常遇到。
- 软件工程数字图书馆对于搜索论文主体，或仅搜索标题、摘要和关键字有着不同的程序。另外，索引系统当然只能用来搜索标题、关键字和摘要。
- 自动化搜索不同的资料源可能会有重叠的结果（也就是说，可能在不同的库中找到相同的论文），而且每次搜索都会包含许多无关的论文[8][13]。

对于手动搜索来说，你需要选择你想要搜索的刊物和会议纪录。当然，你也需要证明你的选择是正确的。然而，在一个案例分析中，我们非常惊讶地发现，有针对性的手动搜索比泛泛的自

动搜索要快许多[32]。在实践中，你很可能需要混合策略。如果你对一些资料进行手动搜索后（包括专家会议的会议纪录），你应该把它当做候选原始研究的一组基准，基于这组基准，你可以验证自动化搜索字符串的效用。另一种方法是，领域专家也可以把一组论文作为确定基准。

医疗标准强制必须有两个研究员独立作出初始的对于包含和排除的决定。然而，我在软件工程中的自动化搜索经验是，许多论文单从标题就能看出它的无关性。因此，在实践中，许多软件工程研究者（包括我自己）允许单个研究者对完全无关的论文作出初始筛选。但此研究者必须理解，如果有任何疑问，那就必须把论文留作候选。然而，在下一步的筛选中，有两个或多个研究者独立评估每篇候选的原始研究是很重要的。在这一阶段，研究者需要完整阅读每篇候选论文，用包含和排除的标准来进行筛选。

如果包含同一项研究多次的话，会使集合产生偏差。因此，必须检查每篇论文来确定他们是否包含多重研究，或者是否因为引用了同样的研究而需要融合他们的结果。然而，要识别出对同一篇研究的重复汇报不总是容易的事。在近期的一次系统性评审中，我和我的同事们必须找出重复的样本量，也要找出重复作者的名单，以便确认对同一篇研究的重复汇报[52]。在其他系统性评审中，问题会更微妙，因为不同的研究者使用同样的数据组来调查同样的数据分析方法，但不总会清楚地指出他们所使用的数据组[31]。

医疗标准要求系统性评审由一个研究团队来执行。这是因为许多流程依赖人的主观判断，特别是：

- 在初始搜索或是运用包含和排除的标准时，决定一篇论文是否成为原始研究的候选；
- 回答质量评估的问题；
- 从每篇原始研究中抽取数据。

让多个研究者来执行系统性评审是希望能削弱个人判断造成的偏差。一开始，我和我的同事们建议采取一个抽取员流程和一个检查员流程，那样也许比两个人独立抽取数据和评估质量效率高[7]。但最后，这被证明是个错误的选择[51]。然而，有时系统性评审必须由单个研究者来完成，比如为研究生学位做研究，或者当人手有限的时候。举个例子，我在时间紧迫和资源有限的情况下，自己完成了一项初步图谱研究[22]。在这种情况下，研究者应该使用一些技巧来评估主观决定的准确性，如Fink所建议的测试–再测试的流程[17]。

最后，关于质量标准，我发现很少有软件工程的系统性评审真正评估了各个单项研究的质量[33][34]。然而，质量判断是系统性评审的流程中不可缺少的元素，它对适当地集合原始研究的结果和解读结果都非常重要。在一次最近针对顺势疗法疗效的系统性评审中，我们可以找到低质量对研究结果影响的例子[46]。如果所有的研究都被包含的话，顺势疗法看起来比安慰剂更有效。但是排除低质量研究的系统性评审清楚地显示了顺势疗法并不比安慰剂更有效。我之后会讨论一个软件工程中利用低质量原始研究的例子。

3.3 软件工程中的系统性评审

在这一章中，我将介绍一些系统性评审的结果，它们对“常识”提出了挑战，而且在某些情

况下，使我被迫改变了一些对软件工程的观念。

3.3.1　成本估算研究

成本估算研究经常汇报对于行业数据集的实证研究，所以有关成本估算的研究话题显然会是系统性评审的候选。确实，一次针对从2004年1月至2007年6月所发布的系统性评审的回顾发现，成本估算的话题是系统性评审中最常见的主题[33]。其中两份评审特别受关注，因为它们颠覆了一些我们对软件成本预估的固定思维。

1. 成本估算模式的准确性

在两份系统性评审中，Magne Jørgensen回答了成本模型估算（从过去项目中收集数据，然后基于数据生成数学公式）是否比专家判断估算（基于软件开发者和管理者的主观意见作出的估计）更准确的问题[21] [22]。自从Boehm[5]和DeMarco[12]在20世纪80年代初出版书籍以来，成本估算的研究者们清晰地表达出了一种信仰：成本估算模型肯定比专家判断更好。但是Jørgensen的评审首次试图确定这种信仰是否有经验性证据所支撑。

在他的研究报告中[21]，Jørgensen找到了15份比较了专家判断模式和成本估算模式的原始研究。他把5份归类为支持专家判断，5份归类为无差别，另5份归类为支持基于模型的预估。在他文后的研究中[22]，他确定了16项比较专家判断估算和正规成本模型的原始研究。他发现，在其中的12篇研究中，专家判断模式的平均准确率显得更高。

这两份系统性评审的区别反映了入选研究的区别和评审侧重点的区别。在更近的那篇系统性评审中，他排除了三篇在第一次系统性评审中所包含的原始研究，并包含了四篇其他的原始研究。选择不同研究的原因是，第一份系统性评审旨在证明改进专家主观估算程序的必要性，第二份系统性评审希望识别出专家意见估算和正规模式估算在何种情况下比另一种更好。然而，尽管两份系统性评审存在区别，结果却清晰地显示成本模式估计不一定比专家判断估计更好。

至于可能存在的论文中“相关研究”章节的偏见，我在此声明，我合著的一篇论文[29]被同时包含在Jørgensen的两篇系统性评审中。在我们的论文中，我和我的同事只确定了Jørgensen所找到的2002年前发布的12篇论文中的两篇。两篇文章都发现专家判断比基于模型的估算更好，这和我们自己发现的一样。

2. 业界成本估算的准确性

1994年Standish Group发布的CHAOS报告[49]显示：软件行业的项目平均超支189%，而且只有16%的项目是成功的（在预算内且按时完成）。Standish Group之后的报告发现超支率有所降低，且有34%的项目成功。这些改变已经使观察者们欢呼雀跃，因为它显示了软件工程技术正在改善。然而，当Moløkken-Østvold和Jørgensen着手对软件工作量估算的研究文献进行评审后[39] [40]发现，CHAOS报告之前已经有三项调查发现了平均超支率为30%～50%。这和CHAOS报告的结果差别很大。因此他们仔细审阅了CHAOS报告，试图理解为什么超支率会那么高。经过调查之后，他们把CHAOS报告从他们的调研中去除掉了。

他们对CHAOS报告的具体调查细节在他们的报告[23]中有所呈现。他们发现了Standish Group使用的方法论有许多不足之处：

- ❑ 没有详细说明计算超支率的方法。Moløkken-Østvold和Jørgensen运行自己的计算方法时，他们发现超支率应该更接近89%，而不是189%；
- ❑ Standish Group似乎有故意寻找失败案例之嫌；
- ❑ 没有低于预算完成项目的分类；
- ❑ 没有良好定义“超支”，“超支”可能包含了已取消的项目的成本。

他们总结说，虽然CHAOS报告是最经常被引用的关于预算超支的论文之一，它的结果却不可信。

3.3.2 敏捷方法

当前，无论是在产业界还是学术界，许多人都热衷于敏捷方法。敏捷方法旨在尽快交付匹配用户需求的应用，并同时确保高质量。许多不同的方法被归在敏捷这一大标题下，但是他们都强调最小化不必要的管理开支（也就是过度的计划和文档），而且聚焦在增量交付特定于客户的功能上。下面介绍一些广为人知的方法。

- 极限编程（XP，XP2）

  初始的XP方法包括12项实践：规划游戏、小发行版、隐喻、简单设计、测试优先、重构、结对编程、集体所有权、持续集成、每周40小时、现场客户和编码规范[3]。修订的XP2方法包含以下“主要实践”：坐到一起、完整团队、富含信息的工作空间、充满活力的工作、结对编程，故事，周循环，季度循环，松弛，10分钟构建，持续集成，测试先行编程，增量设计[4]。

- Scrum[44]

  这种方法关注在难以提前计划的情况下，采用“经验性流程控制（empirical process control）”的机制进行项目管理，关注反馈周期。软件由自组织团队以增量方式开发（被称为“冲刺”），每个冲刺由计划开始，以评审结束。需要实现的系统功能在一份代办列表中注册。然后，产品负责人决定哪项代办列表条目需要在下个冲刺中被开发。团队成员在每天的站立会议中协调他们的工作。有一个团队成员作为Scrum主管，负责解决阻碍团队高效工作的障碍。

- 动态软件开发方法（DSDM）[50]

  这种方法将项目分成三个阶段：项目前期、项目生命周期和项目后期。它基于9个原则：用户必须持续参与，授予团队以权力，注重产品的经常交付，满足现存业务需求，迭代和增量式开发，允许开发过程中的所有变化可逆，在高层次上制定需求的基线，测试自始自终贯穿于开发周期之中，项目所有涉众间的通力合作与沟通。

- 精益软件开发[42]

  这种方法调整了精益生产中的原则（特别是丰田生产系统），使之适应于软件开发。它包含7项原则：消除浪费，增强学习，延迟决策，快速交付，授权团队，嵌入质量，整体优化。

- ❑ 系统性评审可能存在的局限性之一，是它们产生价值的进度可能太慢，以至于跟不上如软件工程这样快节奏的领域。然而，最近已经有两项针对于敏捷方法的系统性评审：Dybå

和DingsøyV的报告[14]，以及Hannay等人的报告[19]。

1. Dybå和Dingsøyr

这些研究者[14]有三个目标：

- 评估当前对敏捷方法的益处和局限性的知识；
- 评估知识背后的证据的强度；
- 在产业界和科研社区应用研究结果。

他们关注敏捷开发的整体，所以排除了调查特定方法的论文，如孤立结对编程的研究。他们确定了33个相关的原始研究，其中大部分研究（24项）调查了专业软件工程师。而另外9项在大学环境下展开。

关于系统性评审和非正式评审的区别，Dybå和Dingsøyr报告称发现了5篇在2003年及以前发表的论文，并未在2004年发布的两篇非正式评审中包含。出于对高质量证据的需求，他们否决了两次非正式评审中所汇报的所有论文，因为它们要么是经验学习的论文，要么就是针对单个实践的研究，而并没有与其他可选方法比较所关注的技巧。

Dybå和Dingsøyr发现，虽然一些研究汇报了XP存在的问题（在大而复杂的项目背景下），大多数讨论XP的论文发现它容易引入而且在不同环境下都运作良好。至于XP的局限性，他们发现有一些原始研究汇报了现场客户这个角色从长期来看不可行。

然而，他们从敏捷方法的实证研究中找出了许多局限性。大部分研究只关心XP，而Scrum和精益软件开发分别只有1篇论文讨论到。而且，只有一个研究小组（被包含的原始研究中有4篇是他们完成的）检测了成熟团队使用敏捷方法的情况。

此外，Dybå和Dingsøyr从不同角度评估了现存证据的质量，包括研究设计的严谨性、单个原始研究的质量（在基本设计的约束下）、不同研究的结果一致程度、以及研究对真实软件开发的代表性。他们发现，证据的总体质量很低，而且他们总结出，除了XP以外的其他敏捷方法需要更多的研究，特别是针对成熟团队的研究，因为研究者可以使用更严谨的方法。

2. Hannay、Dybå、Arisholm和Sjøberg

这些研究者[19]研究了结对编程并开展了一次元分析来集合研究结果。如果你对如何做元分析感兴趣的话，这篇论文提供了可靠的介绍。他们的系统性评审确定了18项原始研究，全部都是实验。18项研究中，4项实验只涉及专业对象，1项同时涉及专业人员和学生，其他13项用学生做研究对象。Hannay等人调研了3种不同的结果：质量、持续时间和工作量（虽然不是每项研究都处理了所有结果）。他们的初始分析显示，使用结对编程有以下效果：

- 对质量的少许正面影响；
- 对持续时间的中等正面效果；
- 对工作量的中等负面效果。

这些结果似乎支持对结对编程影响的标准观点。然而，结果也指出，研究之间有显著的混杂性。混杂性指个体研究从不同的人群出发，因此研究结果不能被很好地理解，除非能识别出不同的人群。

Hannay等人在调查发表偏倚（publication bias）的可能性时发现了更多问题。这指表现出显

著效果的论文比未表现出显著效果的论文更有可能被接受发表。他们的分析指出了发表偏倚的可能性，并发现如果根据可能的偏倚做出调整的话，那对质量的效果就完全抹去了，对持续时间的效果会从中等变成少许，但是对工作量的效果会有一点增加（虽然这仅发生在一项特别的分析模式中）。

他们指出，混杂性和可能存在的发表偏倚可能是由干扰变数（也就是造成不同研究结果之间差异的变数）的存在所导致的。为了研究可能的干扰变数的影响，他们仔细查看了一项研究。那是迄今为止最大的一项研究，涉及295个对象，使用了三个级别的专业软件工程师（高级、中级和初级）。他们总结，任务的复杂性和结果之间可能存在相互作用。所以很复杂的任务可能在使用结对编程时能以大工作量为代价达到高质量，而低复杂度的任务可能以低质量为代价迅速完成。他们推荐研究者在将来的原始研究中关注干扰变数。

Dybå和Dingsøyr的研究显示可以从敏捷方法中获得一些益处[14]。然而，Hannay等人的元分析显示，结对编程的效果可能不如期望的来得强[19]。总体看来，两篇研究的结果都显示出敏捷方法的影响力需要更多的研究，而且我们需要普遍提高原始研究中使用的研究方法。

3.3.3 检验方法

检验技巧是指那些以识别缺陷为目的的阅读代码和文档的方法。他们可能是被研究最多的经验性软件工程的话题之一。在1993年到2002年之间，开展了103项以人为中心的实验[48]，其中37项（36%）研究是关于检验的。其次是面向对象（OO）设计的类别，占据了103篇论文中的8篇（7.8%）。因此，检验方法看似是强有力的系统性评审候选。

最近，Ciolkowski开展了一项包括元分析的系统性评审，来测试关于基于视角阅读（perspective-based reading，PBR）的主张[10]。比如，许多研究者暗示PBR比专案阅读（ad-hoc reading）和基于清单阅读（checklist-based reading，CBR）都要好。而且在一次电子研讨会中，与会专家把影响评估为35%左右[6][47]。而其他研究者对此没那么热情。Wholin等人开展了一次研究调查来定量集合检验方法的可行性，并得出了CBR是最有效技巧的结论[53]。

Ciolkowski的初始元分析发现PBR没有显著影响。结果显示了小到中等的混杂性。一个子集分析显示了PBR比专案阅读强，而CBR在阅读需求文档方面比PBR强，在阅读设计文档方面比PBR弱。他也注明了结果受干扰变数的影响，比如检查材料的出处，以及研究是否由独立研究组所执行。总体说来，使用同样材料和同样研究组的研究相比独立研究更支持PBR，虽然子群分析不太健全。即便如此，他也总结道，PBR能增加35%性能的说法是不被证实的。

3.4 结论

在之前的章节中，我讨论了一些对软件工程提供新见解的系统性评审。在过去几年中，许多其他系统性评审和图谱研究已经涵盖了各方面的话题（详见，Kitchenham等人2009年[33]和2010年[34]的报告，ITS的虚拟特刊①，以及第12章）。我相信这些研究应该能开始改变我们在软件工程

① http://www.elsevier.com/wps/find/P05.cws_home/infsof_vsi_sysreviews。

领域做研究的方法。任何对特定话题感兴趣的人都应该查看针对这一话题的系统性评审。如果系统性评审不存在，也许做一次会是个好主意。如果已有现存的评审，他们可以作为你自己文献搜索的起始点，也可以把你指向那些需要新研究的话题领域。

然而，系统性评审也有显而易见的局限性。一份称为系统性评审的东西不一定保证是一份高质量的评审。在阅读系统性评审的时候，你应该和阅读其他论文一样保持批判的眼光。你可以使用Greenhalgh的评估标准[18]，或者评价与传播中心所使用的5项标准[9]。

- ❑ 评审的选定和排除标准是否有适当的描述？
- ❑ 文献搜索是否涵盖了所有相关研究？
- ❑ 研究是否是合成的？
- ❑ 评审者是否评估了被选定研究的质量和有效性？
- ❑ 基本的数据和研究是否被充分地描述？

系统性评审的第二大问题是对高质量原始研究的依赖。之前章节中讨论的研究显示，虽然原始研究的数量相对较多，但是质量却是个疑问。对于系统性评审来说，我们需要符合如下标准的原始研究：

- ❑ 符合所研究类型的最佳质量的指导方针；
- ❑ 汇报结果时包含充分的细节，以便于元分析的进行；
- ❑ 研究群组和研究材料上相互独立（与Basili等人所建议的实验家族相反[2]）；
- ❑ 收集有关可能的干扰变数的数据，如，研究对象的类型和经验，任务复杂度、大小和持续时间。

此外，即使对以人为中心的方法的原始研究采用了这些最佳实践，我依然坚信除非我们能在现实情况下使用专业的研究对象，基于元分析的集合才能够可靠地评估一项方法和技巧的影响（也就是，完成和现实情况中同样复杂度和持续周期的任务）。

然而，即使原始研究和对元分析结果的解读有些问题，对我来说，实证研究的意义就在于，把实证研究方法和技巧的结果组织进一个经验性知识体。另外，我们需要采取系统性评审的规则来保证我们公平公开地集合我们的成果。

3.5 参考文献

[1] [Antman et al. 1992] Antman, E.M., J. Lau, B. Kupelnick, F. Mosteller, and T.C. Chalmers. 1992. A comparison of results of meta-analysis of randomized controlled trials and recommendations of clinical experts. *Journal of the American Medical Association* 268(2): 240-248.

[2] [Basili et al. 1999] Basili, V.R., F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*. 25(4): 456-473.

[3] [Beck 2000] Beck, K. 2000. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley.

[4] [Beck 2004] Beck, K. 2004. *Extreme Programming Explained: Embrace Change*, Second Edition. Boston: Addison-Wesley.

[5] [Boehm 1981] Boehm, B.W. 1981. *Software engineering economics*. Upper Saddle River, NJ: Prentice-Hall.

[6] [Boehm and Basili 2001] Boehm, B.W. and V.R. Basili. 2001. Software Defect Reduction Top 10 List. *Computer* 31(1): 135-137.

[7] [Brereton 2007] Brereton, O.P., B.A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. 2007. Lessons from applying the systematic literature process within the software engineering domain. *Journal of Systems and Software* 80(4): 571-583.

[8] [Budgen et al. 2008] Budgen, D., B. Kitchenham, S. Charters, M. Turner, P. Brereton, and S. Linkman. 2008. Presenting Software Engineering Results Using Structured Abstracts: A Randomised Experiment. *Empirical Software Engineering* 13(4): 435-468.

[9] [Centre for Reviews and Dissemination 2010] Centre for Reviews and Dissemination. 2007. “DARE” section in the online HELP section. Available at *http://www.york.ac.uk/inst/crd/darefaq.htm.*

[10] [Ciolkowski 2009] Ciolkowski, M. 2009. What Do We Know About Perspective-Based Reading? An Approach for Quantitative Aggregation in Software Engineering. *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement*: 133-144.

[11] [Crowley et al. 1990] Crowley, P., I. Chalmers, and M.J.N.C. Keirse. 1990. The effects of corticosteroid administration: an overview of the evidence from controlled trials. *British Journal of Obstetrics and Gynaecology* 97: 11-25.

[12] [DeMarco 1982] DeMarco, T. 1982. *Controlling Software Projects: Management, measurement, and estimation*. Englewood Cliffs, NJ: Yourdon Press (Prentice-Hall).

[13] [Dieste and Padua 2007] Dieste, O., and A.G. Padua. 2007. Developing Search Strategies for Detecting Relevant Experiments for Systematic Reviews. *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM’07)*: 215-224.

[14] [Dybå and Dingsøyr 2008a] Dybå, T., and T. Dingsøyr. 2008. Empirical studies of agile software development: A systematic review. *Information and Software Technology* 50(9-10): 833-859.

[15] [Dybå and Dingsøyr 2008b] Dybå, T., and T. Dingsøyr. 2008. Strength of Evidence in Systematic Reviews in Software Engineering. *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM’08)*: 178-187.

[16] [Dybå et al. 2005] Dybå, T., B.A. Kitchenham, and M. Jørgensen. 2005. Evidence-based Software Engineering for Practitioners. *IEEE Software* 22(1): 58-65.

[17] [Fink 2005] Fink, A. 2005. *Conducting Research Literature Reviews: From the Internet to Paper*. Thousand Oaks, CA: Sage Publications, Inc.

[18] [Greenhalgh 2000] Greenhalgh, Trisha. 2000. *How to Read a Paper: The Basics of Evidence-Based Medicine.* Hoboken, NJ: BMJ Books.

[19] [Hannay et al. 2009] Hannay, J.E., T. Dybå, E. Arisholm, and D.I.K. Sjøberg. 2009. The effectiveness of pair-programming: A meta-analysis. *Information and Software Technology* 54(7): 1110-1122.

[20] [Higgins and Green 2009] Higgins, J.P.T., and S. Green, ed. 2009. *Cochrane Handbook for Systematic Reviews of Interventions*, Version 5.0.2 [updated September 2009]. The Cochrane Collaboration. Available from *http://www.cochrane-handbook.org*.

[21] [Jørgensen 2004] Jørgensen, M. 2004. A review of studies on expert estimation of software development effort. *Journal of Systems and Software* 70(1-2): 37-60.

[22] [Jørgensen 2007] Jørgensen, M. 2007. Forecasting of software development work effort: Evidence on expert judgement and formal models. *International Journal of Forecasting* 23(3): 449-462.

3

[23] [Jørgensen and Moløkken-Østvold 2006] Jørgensen, M., and K. Moløkken-Østvold. 2006. How large are software cost overruns? A review of the 1994 CHAOS report. *Information and Software Technology* 48: 297-301.

[24] [Jørgensen and Shepperd 2007] Jørgensen, M., and M. Shepperd. 2007. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Transactions on Software Engineering* 33(1): 33-53.

[25] [Khan et al. 2003] Khan, K.S., R. Kunz, J. Kleijnen, and G. Antes. 2003. *Systematic Reviews to Support Evidence-Based Medicine: How to Review and Apply Findings of Healthcare Research*. London: The Royal Society of Medicine Press Ltd.

[26] [Kitchenham 2004] Kitchenham, B. 2004. Procedures for Performing Systematic Reviews.Joint Technical Report, Keele University TR/SE-0401 and NICTA 0400011T.1, July.

[27] [Kitchenham 2010] Kitchenham, B. What's up with software metrics—A preliminary mapping study. *Journal of Systems and Software* 83(1): 37-51.

[28] [Kitchenham and Charters 2007] Kitchenham, B.A., and S.M. Charters. 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering, Version 2.3. EBSE Technical Report EBSE-2007-01, Keele University and Durham University, July.

[29] [Kitchenham et al. 2002] Kitchenham, B., S.L. Pfleeger, B. McColl, and S. Eagan. 2002. An empirical study of maintenance and development accuracy. *Journal of Systems and Software* 64: 57-77.

[30] [Kitchenham et al. 2004] Kitchenham, B.A., T. Dybå, and M. Jørgensen. 2004. Evidence-based Software Engineering. *Proceedings of the 26th International Conference on Software Engineering* (*ICSE'04*): 273-281.

[31] [Kitchenham et al. 2007] Kitchenham, B., E. Mendes, G.H. Travassos. 2007. A Systematic Review of Cross- vs. Within-Company Cost Estimation Studies. *IEEE Transactions on Software Engineering* 33(5): 316-329.

[32] [Kitchenham et al. 2009a] Kitchenham, B., P. Brereton, M. Turner, M. Niazi, S. Linkman, R. Pretorius, and D. Budgen. 2009. The Impact of Limited Search Procedures for Systematic Literature Reviews—An Observer-Participant Case Study. *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement* (*ESEM'09*): 336-345.

[33] [Kitchenham et al. 2009b] Kitchenham, B., O.P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. 2009. Systematic literature reviews in software engineering—A systematic literature review. *Information and Software Technology* 51(1): 7-15.

[34] [Kitchenham et al. 2010a] Kitchenham, B., R. Pretorius, D. Budgen, O.P. Brereton, M. Turner, M. Niazi, and S. Linkman. Systematic Literature Review in Software Engineering—A Tertiary Study. *Information and Software Technology* 52(8): 792-805.

[35] [Kitchenham et al. 2010b] Kitchenham, B., O.P. Brereton, and D. Budgen. 2010. The Educational Value of Mapping Studies of the Software Engineering Literature. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* 1: 589-598.

[36] [Knipschild 1994] Knipschild, P. 1994. Some examples of systematic reviews. *British Medical Journal* 309: 719-721.

[37] [Lipsey and Wilson 2001] Lipsey, M.W., and D.B. Wilson. 2001. *Practical Meta-Analysis: Applied Social Research Methods Series*, Volume 49. Thousand Oaks, CA: Sage Publications, Inc.

[38] [Macdonell et al. pending] Macdonell, S., M. Shepperd, B. Kitchenham, and E. Mendes. How Reliable Are Systematic Reviews in Empirical Software Engineering? Accepted for publication in *IEEE Transactions on Software Engineering*.

[39] [Moløkken-Østvold and Jørgensen 2003] Moløkken-Østvold, K.J., and M. Jørgensen. 2003. A Review of Surveys on Software Effort Estimation. *Proceedings of the 2003 International Symposium on Empirical Software Engineering*: 223.

[40] [Moløkken-Østvold et al. 2004] Moløkken-Østvold, K.J., M. Jørgensen, S.S. Tanilkan, H. Gallis, A.C. Lien, and S.E. Hove. 2004. A Survey on Software Estimation in the Norwegian Industry. *Proceedings of the 10th International Symposium on Software Metrics*: 208-219.

[41] [Petticrew and Roberts 2006] Petticrew, M., and H. Roberts. 2006. *Systematic Reviews in the Social Sciences: A Practical Guide*. Malden, MA: Blackwell Publishing.

[42] [Poppendieck and Poppendieck 2003] Poppendieck, M., and T. Poppendieck. 2003. *Lean Software Development: An Agile Toolkit*. Boston: Addison-Wesley.

[43] [Roberts and Dalziel 2006] Roberts, D., and S. Dalziel. 2006. Antenatal corticosteroids for accelerating fetal lung maturation for women at risk of preterm birth. *Cochrane Database of Systematic Reviews* 2006, Issue 3. Art. No.: CD004454. Available at *http://www2.cochrane.org/ reviews/CD004454.pdf.*

[44] [Schwaber and Beedle 2001] Schwaber, K., and M. Beedle. 2001. *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall.

[45] [Shadish 1995] Shadish, W. 1995. Author judgements about work they cite: Three studies from psychological journals. *Social Studies of Science* 25: 477-498.

[46] [Shang et al. 2005] Shang, A., K. Huwiler-Müntener, L. Nartney, P. Jüni, S. Dörig, D. Pwesner, and M. Egger. 2005. Are the clinical effects of homeopathy placebo effects? Comparative study of placebo-controlled trials of homeopathy and allopathy. *Lancet* 366(9487): 726-732.

[47] [Shull et al. 2002] Shull, F., V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelowitz. 2002. What We Have Learned About Fighting Defects. *Proc 8th IEEE Symposium on Software Metrics*: 249-258.

[48] [Sjøberg et al. 2005] Sjøberg, D.I.K., J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.K. Liborg, and A.C. Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering* 31(9): 733-753.

[49] [The Standish Group 2003] The Standish Group. 2003. Chaos Chronicles Version 3.0. Available at *http://www.standishgroup.com/chaos/intro1.php.*

[50] [Stapleton 2003] Stapleton, J. 2003. *DSDM: Business Focused Development*, Second Edition. Upper Saddle River, NJ: Pearson Education.

[51] [Turner et al. 2008] Turner, M., B. Kitchenham, D. Budgen, and P. Brereton. 2008. Lessons Learnt Undertaking a Large-scale Systematic Literature Review. *Proceedings of EASE 2008*, BCSeWiC. Available at *http://www.bcs.org/upload/pdf/ewic_ea08_paper12.pdf.*

[52] [Turner et al. 2010] Turner, M., B. Kitchenham, P. Brereton, S. Charters, and D. Budgen. 2009. Does the Technology Acceptance Model Predict Actual Use? A Systematic Literature Review. *Information and Systems Technology* 52(5): 463-476.

[53] [Wholin et al. 2003] Wholin, C., H. Petersson, and A. Aurum. 2003. Combining data from reading experiments in software inspections: A feasibility study. In *Lecture Notes on Empirical Software Engineering*, ed. N. Juristo and A. Moreno, 85-132. Singapore: World Scientific Publishing.

3

第4章

用定性研究方法来理解软件工程学

Andrew Ko

人们总是相信数字。数字是计算机学的核心，金融学的基础，也是过去一个世纪中人类进步的重要组成部分。对于现代社会中的大部分人来说，数字还是认识事物的方法：我们使用数字来研究药物的安全性，也用它来研究哪种政策比较好，还用它来研究宇宙的演变。数字也许是人类最强大最有效的工具之一。

但是和所有的工具一样，数字也有其弱点。有的问题用数字就很难回答。比如说，20世纪80年代的公共卫生研究员们曾想找出为什么癫痫病人很难按时吃药。他们计算了未能按时吃药的人数，并用统计学的方法找出了按时和不按时吃药的病人的区别，甚至进行了超长期对比研究实验，以找出不按时吃药的后果。但是这些方法并不能解释病人不能按时吃药的原因——它们只能用严谨的口吻来描述一些明摆着的事实。

后来，另一群研究员用完全不同的方式进行了一次革命性的研究[1]。研究员们没有去做计算和统计，而是对癫痫病人进行了80次访谈，询问这些受访者在什么时候会忘记吃药。最后他们发现不按时吃药的原因并不是因为病人不理智或者反复无常，而是因为病人自己故意不吃。比如，有的病人明白这些药物可能会让人产生依赖性，所以就有意识地节制了药物的使用以免产生抗药性。在公共卫生的领域中，这样的人文研究在当时是头一遭，而研究的结果也极大地推动了癫痫病症给药方式的改革。

这些和软件开发有啥关系？和公共卫生一样，软件开发也充满了很多“为什么”和“怎么办”的问题，这些问题无法用数字和统计来解决。比如，如果你是一个软件开发团队的管理者，你可能常会问自己一些问题。为什么我的开发人员不愿意写单元测试？为什么用户总是填错这个表单？为什么有的开发人员的效率是其他人的10倍？这些问题无法用数字来回答，但是如果我们能小心地运用*定性研究*的方法，就可以回答它们。

但是定性研究方法并不是说简单地问人们一些问题就行了，阅读定性研究报告也不像阅读研究摘要那么简单。本章将介绍什么是定性研究方法，如何来解读定性研究的结果，以及如何在自己的工作中应用这些方法来提升软件开发流程以及软件的质量。

4.1　何为定性研究方法

简单地说，定性研究方法指系统地收集和解读非数字的数据（包括文字、图片等）。和定量研究方法一样，定性研究方法可以收集数据来证实或者推翻一个观念（使用演绎推理法）。此外，定性方法还可以用于支持归纳推理法：收集数据以便能提出新的解释。由于定性研究法收集的是非数字的数据，所以这种方法的使用场景也更加自然。

为了解释这一点，让我们看看一个案例，并讨论如何使用定性研究的方法来理解它。想象你刚刚接管了一个软件开发团队。你现在正领导团队进行一个新项目，而这个项目刚刚配备了一个新的bug跟踪系统，包含了很多你的团队想疯了的新功能。在接下来的几个月里，你看到你的团队不停地对代码修修补补，进度喜人，但是每次你检查新系统中的bug列表时，却只有少量报告存在，它们都是由少数几个测试人员提交的，而且似乎始终是这几个人。后来有一天早上，你走到你手下最好的开发人员的办公桌前，却发现他的屏幕上显示的是旧的bug追踪系统。你的团队原来一直在暗中使用旧系统！有了全面的培训和小心的过渡，而且软件的许可费用也很贵，为什么开发人员们不用新系统呢？

很明显我们可以直接问。比如，你可以召集一个会议，并询问团队为什么不用新系统。你可能会得到一些解释，但是这些解释未必完全可信，尤其是在开发人员发现他们自己正在忤逆你的意见的时候。此外，由于他们处在特定社会环境中，所以比较内向的成员可能就不会开口说话，那么你得到的解释就只是那些说话最多的成员的一面之辞。勉强而又不全面的解释没有任何用处。

要想跳出这个圈套，你可以让一个你信任的朋友在休息时间四处打听，简短地非正式采访一下这些开发人员。这样开发人员们都不必当着大家的面来表达自己的想法，说出来的话就可能更多，也更诚恳。但是局限性仍然存在，得出的观点仍然局限于你朋友的朋友的意见。更惨的是，你的朋友本身就可能不公正。可能他自己就很讨厌新的系统，并在报告给你的时候不自觉地偏向于一边。

刚刚我所说的方法的问题在于，得到的数据可能是二手甚至是三手的。在理想的情况下，你应该能自己观察到最有意义的时刻。比如，当开发人员决定使用旧系统而不是新系统的时候，他们在想些什么？他们有哪些时间上的压力？他们在和哪些人合作？他们输入了哪些数据？为了找到答案，你可能需要在开发人员身边坐一整天，看看他们在做什么。这样你就能直接观察到他们决定放弃新系统，使用旧系统的那一瞬间。

当然，想要直接观察到开发人员的这些瞬间通常是不现实的。人不喜欢被监视，而且经常会调整自己的行为以保护隐私、避免尴尬或者解除焦虑情绪。此外，由于你是唯一的观察者，你很可能会把你自己的偏见也带进来。

我们还可以把开发人员从这个方法中完全剔除出来，只研究文档。开发人员们在旧系统中报告了哪些bug，又在新系统中报告了哪些bug？这样我们也许可以发现开发人员不愿在新系统中报告的bug类型。这样你也许可以找到一些使用旧系统的原因，但是真正的原因仍然存在于团队成员的头脑之中。

上述几种都是定性研究的方法，他们都有各自的局限性。要想解决这些限制，我们就必须接受它们：只用一个方法无法揭示完整而不偏不倚的事实。好的人文研究会结合多种方法，并将各种方法得出的证据结合在一起。举例来说，比如你既对员工们进行了访谈，又调查了哪些bug是在新系统中提交的。两种方法给你的结论虽然不同，但相互之间或多或少的有一致性。通过对比这些结论的相同点和不同点，你就可以了解到这个问题的真正答案。

除了调查了解事实之外，定性调查还可以让我们进行换位思考。很多时候，沟通不畅、违反流程、士气低落都是因为人无法从别人的角度来看世界。而这也正是定性研究想要解决的问题。这种换位思考往往就是定性研究方法的目标。比如说，你管理着一个团队，发现每周五都会出现编译错误，人们整天都在解决这个问题，每个人回家的时候都很累。使用定性研究的方法来调查这个问题你可能会发现原来星期四晚上大家都在忙着提交修改，因为周五是你决定开会的日子。有了这样的调查方法，你就可以从开发人员的角度来看这个世界并找到让大家都满意的解决方案。

4.2　如何解读定性研究

在了解了什么是定性研究方法之后，我们现在将介绍一下如何来解读定性研究（包括这本书中将出现的各种定性研究的成果）。我们将涉及一些问题，比如这个研究让我们学到了什么？在什么情况下你才能相信这些研究的结论？在什么情况下你可以把这个研究的结论推广应用到更多的项目中？为了解答这些问题，我们可以参考一下图灵奖获得者Donald Knuth于1989年发表的文章“TeX的错误”（*The Errors of TeX*）[2]。

在这篇经典的文章中，Knuth分析了他在编写TeX软件的过程中所记录下的超过850项错误。按Knuth的说法，这篇文章的目的是“展示在TeX的开发过程中修复过的错误，并尝试分析这些错误”。Knuth阐述他使用这个方法的初衷是为了解决量化研究的一些局限：

> 光用数字是无法表示出规模的概念的。我相信一个详细的列表可以给我们很多重要的启示，而这些启示是无法通过统计数据获得的。

Knuth的发现是什么？他从一个大型的错误数据库中整理、展示并描述了15种错误，还分别为每种错误附上了例子。比如说“粗心和失误”类错误，就包括了语法正确但是语义错误的程序语句。这些错误的根本原因是有的变量名字的概念紧密相关，但是却会导致非常不同的程序语义（比如表示前、后变量的名字调换，或者比如*next_line*和*new_line*等）。Knuth还介绍了其他的错误种类、TeX项目的历史、他写TeX时的一些经历以及他记录错误的方法。

在文章的最后，他总结道：

> 那么我到底学到了些什么？我认为主要的收获是我的平衡感和比例感更好了。现在，我明白了一个中等规模的软件系统的复杂性之所在，以及这个系统的发展方向。我也明白了错误的种类是如此繁多，所以我们无法用系统的方法来去除一切可能“被认为是有害的”东西。我还明白了我有犯错误的倾向并接受现实，然后当我犯错的时候可以更积极地承认。

让我们返回上一步，重新思考一下这篇文章的价值：作为读者，我们又学到了些什么？我在20世纪90年代中期的时候第一次读到了这篇文档，当时我学到了很多：我从来没有编写过中等规模的软件系统，而文章中详细的背景以及历史信息让我学习到了独立开发这样一个大型系统的经验。我发现Knuth介绍的很多错误类型在我自己编程的时候也有，不过也发现了很多不曾见过的问题，这些都让我可以更好地思考我的代码出问题的原因。作为一个研究员，我还从这篇文章中学习到了软件开发背后的人为因素，即：我们如何思考，我们的记忆是如何工作的以及我们如何做计划以及进行推理。这些都是影响软件质量的重要因素。和这篇文章类似的还有一些，它们把我引领到了一条研究人为因素的职业道路上，让我可以通过研究更好的编程语言、工具以及流程来改善软件的质量。

不过，在阅读之后马上就学到东西的情况并不多见。我也是在读完之后的几个月中逐渐在自己的工作中意识到Knuth所描述的错误种类，而除了这篇文章之外，还有很多文章都引起了我的研究兴趣。这是精细地解读定性研究的重中之重：研究的推论需要时间才能体现出来，而且你还需要坦诚地对这些推论进行反思。如果你仅仅因为某个纰漏或者你不认同的结论就把整篇文章都作废，那么你可能会错过所有其他有用的启示，而想要得到这些启示你必须要认真而持续的对研究的结果进行反思。

当然，这并不是说你就应该对Knuth的结果全盘接受。只是，你不应该感情用事地对待研究的结果，而应该用更加系统的方式来进行解读。一般来说我主要关注研究的三个层面：*输入*、*执行*和*输出*。（听起来就像软件测试，不是吗？）我就以Knuth的研究为例来讨论一下这几个层面。

首先，你信任Knuth的研究的输入吗？比如说，你认为TeX是一个典型的程序吗？你认为Knuth是一个典型的程序员吗？你信任Knuth本人吗？上面这些因素可能会影响你对研究的看法，比如：Knuth的15种分类是否全面，是否典型，在数十年之后是否仍然会在工作中发生。假如你认为Knuth无法代表其他的程序员，那么要是换成别人来做这个研究，结果会有怎样的不同呢？比如说，我们可以假设Knuth和很多的学界人士一样，是个心不在焉的教授。也许这可以解释为什么有那么多的分类都和遗忘什么事情或者缺乏远见有关了（比如分类包括了*被忘记的功能*、*模块之间的不匹配*以及*意料之外的情况*等）。也许一个更加自律的人或者一个只关注代码的人就不会有那么多问题。这些干扰因素并不能否定研究的结果，但是在你将这些结果推广应用之前，还是应该仔细考虑它们的影响。

你是否信任Knuth的研究的*执行*？换句话说，Knuth有没有严格按照自己描述的方法来执行实验，以及如果他没有的话，这些偏差又会对研究结果造成什么影响？Knuth使用了*日记研究法*，这种方法在今天常用于调查人在长时间内的经历和感受，而无需直接对其进行观察。要想用好日记研究法，有一个关键点就是不能告诉参与者你的实验想要找出什么结果，以免这些暗示让他们产生偏见，影响了他们写什么和如何写。但是在这份研究中，Knuth既是研究员，又是参与者。他希望能得到什么样的结果？是不是在写日记之前他在脑海中就已经分好了类？他是在TeX的开发过程中为这些错误分类，还是在TeX开发完成后来回顾分类的？Knuth并没有在研究报告中介绍这些细节，但是这些细节却可能大大地改变我们对研究结果的解读。

日记研究也有其局限性。比如说，这种研究有可能会造成海森堡式的问题，即观察的流程可

能会强迫写日记的人不断地对记下来的问题进行反省，以至于最后连记录的性质也变了。这意味着由于Knuth一直在记录各种错误，他可能会认真地思考造成这些错误的原因，最后有意识地避免了各种错误，最后就无法观察到所有结果。此外，日记研究还有一个难题，就是参与者很难长时间地坚持。比如说，Knuth有段时间曾暂停了他的实验，说道："我没有记录下在Tex82开始紧张的调试工作中所解决的错误……"如果Knuth在那段时间做了记录，他会发现什么样的错误？这些错误和在平时压力不那么大，不那么紧张的时期发现的错误是否会不同？

最后，你是否信任这项研究的输出及其推论？学术论文的写作标准要求将研究结果和推论区分开来，以方便读者确定他们是否可以根据同样的证据得出同样的结论。但是在Knuth的文章中，他从头到尾都把两者混在了一起，不但详细地介绍了TeX的各种故障，也提出了他的研究推论。比如说，在描述了关于意外类错误（Knuth说这些是全局误解）的各种有趣故事之后，他这样说道：

> 这段经历告诉我们，所有的软件系统都应该经历你能想象到的最不友善最难对付的测试的磨炼，否则，即便在大量部署并取得较好的反响之后的数年中，我们也几乎可以肯定系统会不断地产生bug。

当结果和推论并排在一起的时候，读者就很容易忘记这二者的区别，也会忘记应该将它们分开来分析。我相信Knuth对于这些（导致了上述推论的）故事的记忆，因为他描述了他记录这些故事的流程。但是，我认为Knuth过度解读了这些故事的意义，使其更符合他的结论。如果Knuth花了如此多的时间在痛苦测试上，他还有可能完成TeX吗？我相信他的日记，但是对他的最终建议，我保留怀疑态度。

当然，我们有必要重申一点，那就是每个定性研究都有其局限性，但是大部分的研究都能给我们宝贵的启示。作为一个定性研究的客观解读者，我们必须接受这个现实，并认真地辨别这两点。好的研究报告会帮你解决这个问题，比如这本书里介绍的研究。

4.3　在工作中运用定性研究方法

虽然定性研究方法通常是在学术研究中使用，但是这些方法在工作中也有很大的作用。实际上，当你需要回答任何没有标准答案的问题的时候，这些方法就可以帮上忙。而在软件工程中遇到的问题几乎总是如此。软件测试人员可以使用定性研究方法来解决在测试中遇到的与人相关的低效问题，而项目经理们则可以使用这些方法来研究团队的社交指标会如何影响效率。设计师、开发人员以及需求工程师们可以使用定性研究方法来更深入地理解他们的用户，以便更好地满足用户的需求。当你需要分析谁、什么、何时、何地、怎么做以及为什么之间的关系的时候，定性研究方法就可以帮上忙。

使用定性研究方法就和当个名侦探或者记者一样：关键是要发现事实真相，并讲一个令人信服的故事——但同时还必须意识到从不同的角度会看到不同的事实真相。你会如何来发现这些角度在很大程度上取决于研究的状况。你需要对研究所处的社会背景有所领悟，这样你才能知道谁可以信任、他们存在哪些偏见以及他们的哪些动机可能会影响你的判断。只有具备了这些认识之

后，你才能确定如何结合使用直接观察、跟踪观察、访谈、文档分析、日记研究等方法。

为了说明这一点，我们不妨回到bug跟踪系统的例子。你是选择研究方法的最重要的因素，所以你必须考虑这样一些问题。

- ❑ 你的员工是否喜欢你？
- ❑ 他们是否尊敬你？
- ❑ 他们是否信任你？

如果任何一个问题的回答是“否”的话，也许你就不适合做这个研究。相反，你可能需要找到一个更加公正的合作伙伴来替你进行这个研究。举例来说，*监察员*就是非常好的研究人员的候选。他们是中立的一方，但是他们又必须从多个角度看待问题，这样才能更高效地沟通和解决问题。如果在你的公司中有监察员，那么他们将是非常好的学习定性研究方法的候选研究员，而且还将在改善工作环境的过程中扮演重要的角色。

如果你的员工确实很喜欢你，那么另一个重要的因素就是你的员工。他们能否很好地进行沟通？他们是否诚实？他们是否喜欢隐瞒不报？你的团队有没有分派系？在团队中有没有不断改善的传统，或者说团队是否僵化而保守？这些都是确定某种研究方法是否可行的社会因素。举例来说，如果员工很喜欢隐瞒不报，那么就不能使用直接观察的方法，因为他们知道自己正被观察。文档分析可能会有用，但是你的团队会不会考虑隐私的问题？访谈只有在访问者能与受访者建立融洽关系的时候才有效，否则就得不到任何效果。好的定性研究方法的目标是找到一种观察的方法，既能使偏见最小化，又能使客观性最大化。

无论是你还是别人来做这个研究，还有一个重要的考量就是你如何来向团队解释这些研究。在任何情况之下，“秘密进行研究”的想法都是极坏的。你提出问题的原因也许是因为你想解决问题，而你正管理着一整个团队的人，他们都可以解决问题……那还不让他们来帮忙！不过，你必须明确告诉他们，你的目标是了解他们，不是命令他们。你还应该表明每个人的解释不同很正常，因为每个人的角度不同。在表明了这些之后，团队成员们就会明白你想从不同角度来看问题的愿望，并和他们产生共鸣，这样他们就更容易真诚地进行反馈。

最后，定性研究方法有时候会让人感觉松散而随意。你怎么能相信这样一个没有结构的流程所得出的结果呢？问题的关键在于你必须认识到自己周围的偏见并接受它。作为研究员的你有偏见，你的受访者有偏见，而你所使用的各种研究方法也都偏向于揭示某种现象。你越能明确地识别这些偏差，越了解它们会如何影响你对研究结果的解读，你的研究结果就越客观。

4.4 推广应用定性研究的结果

无论你是在论文中看到定性研究的结果还是你自己做的研究，都会有一个问题，那就是这些结果能在多大程度上被推广到别处？

比如说，定性研究可以找出*常见*（common）的行为，但是他们无法识别*普遍*（average）的行为。普遍（即平均）的行为以及其他汇总统计都需要计算才能得出，而定性研究和计算无关。更重要的是，由于收集非数字的数据需要更多的人力，定性研究的方法通常只能得到较小的样本

量，这样想要推广应用研究的结果就会很困难。

不过，定性研究的结果是可以推广应用的。比如说，有些研究可以证明在一个环境下出现的因果关系和另一个环境下的因果关系类似。比如说你是某个Web程序的UI团队的成员，想要了解为什么代码审查总是花那么长的时间。虽然你的研究结论可能无法推广到管理数据库后台的团队，但是却可能推广到其他类似领域的Web程序的前端团队。正确地认识在什么时候可以推广研究结果和定性研究本身一样困难。二者都需要依靠主观判断来确定哪些假设在新的环境之下仍然有效。

4.5 定性研究方法是系统的研究方法

在软件工程的研究中定性研究方法的使用越来越多，而对于软件工程实践来说，这些方法也很重要。但是简单地认识问题和*系统地*认识问题有着很大的区别。所以，下次你阅读定性研究的结果或者自己进行研究的时候，你需要确保你（或者你的文章）遵循下面这样的流程。

- 系统地表达问题

  我们需要了解什么？为什么？我们能拿了解到的知识来做什么？

- 考虑可行的客观证据来源

  哪些数据可以*客观地*收集，以尽量地减少偏见？你能不能从多个来源收集信息，以解决单一来源的偏见问题？

- 解读证据中出现的模式

  这些证据来源相互之间是一致，还是相互冲突？这些证据导致了什么新问题？这些问题能不能形成一个新的假说，并用更多数据来证实？

虽然这样一个表达问题、收集证据、解读证据的周期可能会很长，但是每通过一轮，你都将对你的认识更有信心，这也意味着可以做出更好的决策，以及得出更好的结果。此外，结合定性研究方法和定量研究方法，你就能得到一套完善而强大的研究工具，有了这些工具的帮助，你将可以更有效地理解和改进软件工程实践。

4.6 参考文献

[1] [Conrad 1985] Conrad, P. 1985. The meaning of medications: Another look at compliance. *Social Science and Medicine* 20: 29-37.

[2] [Knuth 1989] Knuth, D. 1989. The Errors of TeX. *Software Practice and Experience* 19(7): 607-685.

第5章

在实践中学习成长：软件工程实验室中的质量改进范式

Victor R.Basili

实证研究，即使用广为认可并且经过验证的方法以证明某种断言真实性的正式研究方法，正逐渐在软件工程中取得进展。令人欣慰的是，人们终于承认了实证研究在软件工程科学中的重要地位。我们也在科学文献中看到越来越多的实证研究和相关实验，它们能够证明或者否定某些方法、技术和工具的有效性。

但可惜的是，实证研究仍未被用于新的探索发现，只是在概念完善后锦上添花而已。但从传统上来说，实证研究这种科学方法是指运用某种方法、技术或工具，分析结果，从而改进先前的理念。也就是人们一直以来检验理论并对它们进行改进的方法。在软件工程中，各种理论与模型仍然处于形成阶段，而人们一直将这一过程当做创造性过程的一部分来加以应用，所以观察实践以及进行探索性研究对于该学科的成长至关重要。

5.1 软件工程研究独有的困难之处

软件工程与其他学科有几点不同之处。首先软件是人类智力与创造力的产物，而不是机械地“制造”出来的。软件开发的流程也不同于制造的流程。换而言之，软件流程并不是一种简单的重复劳动。这是该学科不同于其他学科的关键之处，也极大地影响了我们对它的解读。我们时时刻刻都需要关注环境变量所带来的影响。由于软件工程是基于“人”的学科，所以研究的结果总会有差异，我们也无法控制甚至不能确认所有的环境变量。所以在探索如何为人们修改和调整流程的过程中，我们需要不断进行实验。

环境差异并不单纯指软件开发人员的不同。软件的每一部分都不尽相同，软件开发环境亦是如此。这也就意味着，流程是变量，目标是变量等。我们需要为所研究的环境选择合适的目标和合适的流程。所以，在决定如何研究某种技术之前，我们需要对其环境和特性做一些了解。

软件工程的第二点与众不同之处在于软件的*无形性*，也就是构架、组件、开发形式的不可见。这一点和软件工程的第三个特性，即该领域的*不成熟性*密切相关，因为仍没有完善的模型能帮助

我们推导流程、产品和它们之间的关系。这些困难促使以下两种需求变得尤为迫切：从不同情境下的理念应用中学习，并且从中归纳理论或者模型。

最后一点在于，把经验建立成可重用的模型需要额外的人力、物力、流程和组织的支持。建立模型、进行度量、利用实验找出最有效的技术，并生成反馈信息用来共同学习都需要时间和金钱的投入。这些活动并不是软件开发的副产品。如果没有独立于产品开发并且明确支持这些活动的方法，它们就不会发生，我们也就无法从根本上改进开发流程。

由此可见，成功的实验是困难而昂贵的。实验的结果只能在小范围内确证有效，而受制于大规模适用性、研究方法整合以及环境变量等问题。

5.2　实证研究的现实之路

我相信我们有必要关注非正式的探索性研究，并从中得到关于软件开发的指导意见。在适当的时候，将这些意见与正式的实证研究相结合来验证我们的理论，这些理论会加入整个理论体系并最终使软件工程更加明了。所以我认为软件工程学科的研究是探索并逐步进化的。软件工程研究遵循科学方法，但由于其特性，真实实验并不总是可行而有效的。

在我看来，软件工程研究是一门实验室科学，而且是一门内容广泛的大学科。实验室的规模之大，使得我们必须使用更具探索性的方法来进行研究。只通过分析是无法理解这门学科的。唯有通过在软件工程的实践中研究各种关系的产生和变化，以及各种技术的限制，我们才能知道如何通过配置流程优化软件开发。

同时我们需要利用一切实践机会探索各种理念，比如对可行性进行测试，看看人们是否可以应用这些理念，理解应用它们所需要的技术，并测试它们与其他概念的相互关系。基于这些信息，我们可以调整研究环境中每一个理念，使人们可以在实践中轻松使用。甚至在建立模型之前，我们就需要在实践中试用我们的理念并且改良它们。简而言之我们从事的是探索性研究，我们比很多科学都更多地依赖于方法和技巧的实证应用。

多样性是我们在软件工程中运用科学方法时的一个重要方面。对于一种流程，我们需要验证它在不同的环境下的表现，才能更好地理解流程背后的理念以及它们之间的相互作用。随着时间的推移，我们还需要考虑所有的环境变量。如果没有在不同地点、由不同人对流程进行应用的话，很多环境变量的影响根本无法体现出来。

仅靠一个团队绝不足以建立这样的理论体系。作为一门庞大的科学，它需要依靠团队之间的合作。研究结果也需要被高效地共享，比如可以建立一个存放不断完善中的模型以及经验教训的知识库，研究人员可以使用和更新其中的内容。

5.3　NASA 软件工程实验室：一个充满活力的实证研究测试平台

为了支持我在本章提到的非正式学习的观点，我将总结我们在NASA（National Aeronautics and Space Administration，美国宇航局）软件工程实验室（SEL）25年的经验。在SEL，我们不

仅通过实验来学习，而且通过分析问题、应用可行的解决方法、并找出其不足之处来不断学习经验。我们也进行了对照实验与案例研究，但是是在一个不断改进的更大型的学习流程中执行的。

SEL建立于1976年，旨在为卫星系统的软件开发提供地面支持，并且在可能的情况下用观察、实验、学习和建立模型的方法为位于Goddard①的飞行动力学部门优化流程和提高产品质量[3]。实验室的研究团队由NASA与CSC电脑科技集团（CSC）的开发人员，以及马里兰大学的一个研究团队组成，三者组成了一个联盟。SEL决定支持我们的实证研究，并且将其并入组织的整体工作中。实验支出来自项目自身的经费，而非来自NASA的研究经费。

在1976年，几乎没有关于软件工程的实证研究，而建立一个研究软件开发的实验室更是前所未有的。该实验室的建立提供了一个良好的学习环境，在实验室中，人们设计并运用潜在的解决方案、检验其有效性、从中总结经验，从而优化解决方案。实验室活动只限于一个特定领域，由专业的开发人员参与，同时得到当地机构的鼎力支持。研究团队与实际开发人员紧密交流，这是一个开发人员和管理人员组成的团队，他们性格各异，拥有不同的职责和目标。以上这些特质使其成为进行实证研究的沃土。实验室各方面的平衡创造了一个理想的学习环境，大家通过彼此合作及反馈可以相互学习。

从1976年到2001年，我们在不断犯错中学到了很多。所犯下的错误包括：

- 在全面认识环境之前就妄下断言；
- 是数据主导，而非目标主导或是模型主导；
- 用其他环境下得出的模型解释我们的环境。

学习的进化流程是渐进的而非跃进的。利用每次学习的经历，我们试图把学习到的知识运用到流程模型，产品与组织机构上。

SEL请大学研究人员验证高风险的方案。我们建立模型并测试假设。同时我们开发技术、模型和理论来解决问题，从而了解哪些方案是有用的。我们在应用中借鉴他人的方案，也研究出一些自己的方案。实验室的研究就是如此进行的。

我们重要的发现是如何把科学方法运用于软件领域，即如何从概念运用、案例研究、以及对照实验的反馈信息中得到启发从而改进特定环境中的软件开发流程。非正式的反馈指导我们应该关注案例研究与实验中的哪些部分，同时我们也出乎意料地从非正式的反馈中得到了关键的启发。

本章回顾了我们实际运用科学方法的实践流程，以及我们是如何基于实践中得到的反馈而改进方法的。

5.4 质量改进范式

我将从最精华的部分说起，即在商业环境下把科学方法运用于软件工程的过程。我们把这种科学方法称为质量改进范式（QIP）[2] [7]。它一共分成6个基本步骤。

① Goddard空间飞行器中心，NASA主要的空间研究实验室之一，位于美国马里兰州。——译者注

(1) 使用适当的模型和指标表征手头的项目与环境。（我们的世界是什么样子的？）

(2) 为项目的成功与进步设定量化的目标。（对于世界我们想知道些什么，我们要达成什么目标？）

(3) 为该项目选择流程模型，以及支持该流程模型的方法与工具。（在这样的环境下，需要运用怎样的流程来实现目标？）

(4) 执行流程，搭建产品，搜集、验证、分析数据为修正提供实时反馈。（执行选定的流程时发生了些什么？）

(5) 分析数据以评估现状、确定问题、记录发现、为将来的项目做出改进意见。（所采纳的方案表现如何？缺少了什么？如何解决？）

(6) 封装得到的经验用来改进模型，把当前项目与以往项目中得到的知识结构化，存入经验库中以便重用。（如何把经验为机构所使用？）

质量改进范式（QIP）是一个双循环的流程，如图5-1所示。研究与实践进行交互，二者分别由理念应用中所得到的项目经验和共同经验所代表。

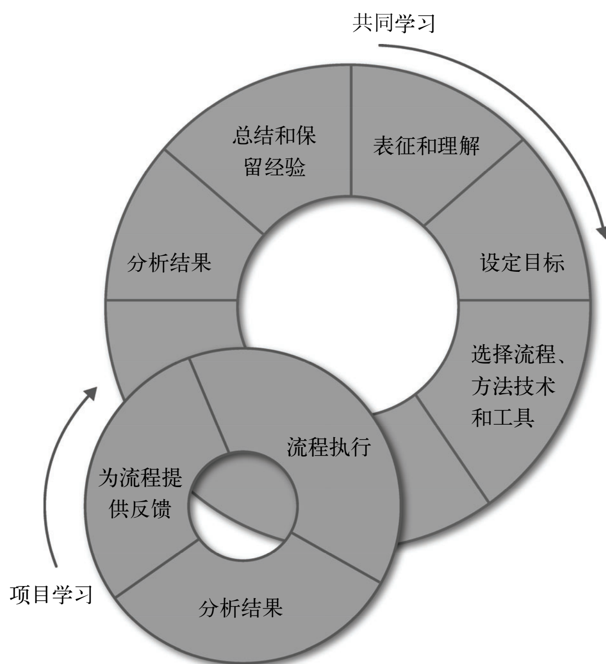


图5-1　质量改进范式

然而QIP并非一开始就是如此。随着对执行流程的观察，我们久而久之用所得到的知识改进了每个步骤之后最终形成了这个方法。之后我将会讨论步骤的改进以及得到的启发，在这里，正式实验对主要理念的证明只起到支持的作用。

整个学习流程跨越25个年头，本章中我将集中介绍前20年的研究所得。虽然它们之间存在角

色重叠，我的讨论将覆盖QIP中的6个步骤。对于每一个案例，我会介绍我们在应用科学方法时得到的经验教训，以及我们在NASA/GSFC的飞行动力学部门中改进软件开发时得到的经验。

从1976年起，我们进行了如下一些活动，对应于质量改进范式的6个步骤：表征、设立目标、选择流程、执行流程、分析、封装。

5.4.1 表征

起初，我们试图使用已经存在的一些模型来帮助我们理解环境（比如雷利波曲线或是MTTF模型）。结果我们发现它们并不合适，要么因为它们的定义是服务于相对更大型的项目（以100KLOC为单位），要么就是使用它们的时间点并不符合我们的需要。所以我们开始用自己的数据设计适合于自身环境的模型。我们需要更好地理解和表征环境、项目、流程等，因为我们无法使用别人从其他环境中提取的模型[14] [6] [5]。

我们必须了解我们的问题区域。久而久之，我们开始建立基线来帮助我们了解局部环境。在图5-2中，每一个框表示对应某一个卫星的地面支持软件。我们为此建立了成本的基线、缺陷的基线、重用百分比的基线、缺陷类型的基线、工作量分布的基线、代码库中代码增长的基线等，并用这些基线信息定义目标，形成历史数据做为比对的基础。

在不断改进的流程中，我们发现需要更好地理解那些使得项目相同或者不同的因素，这样可以帮助我们选择更适合的模型，并且知道哪些变量影响了项目的效率。背景信息，甚至是局部环境之内的也是十分重要的。

5.4.2 设立目标

从一开始我们就决定使用度量作为抽象化方法以使发生的行为清晰可见，并且建立数据采集表格与度量工具。我们搜集了6个项目的数据并存入一个小型数据库，并试图解释这些数据，结果发现有些时候没有正确的数据或是充足的信息来回答我们的问题。所以我们意识到不能单纯地搜集数据然后再计划如何处理它们；数据的采集必须是以目标为导向的。这促使我们编写了目标问题矩阵方法（GQM），用它指导对一个特定研究的数据采集工作[12]。但这样仍然可能会引入过多的数据，特别是没有目标的时候。我们不断地改进GQM，比如定义目标模板[10]。

我们还认识到使用基数定量与序数标量可以在结研究果中捕捉到其他方式难以得到的信息。随着模型的发展，根据该环境中超过100个项目的经验，我们把数据库也换成了基于模型的经验库。

5.4.3 选择流程

我们首先探索性地重组了现有的流程，尽可能降低对流程的影响。然后我们开始在大学中与学生一起做对照试验，以便用最小成本区分出某几个变量的影响[9]。在理解了这些流程的效果之后，我们就开始用成熟且具有影响力的技术集来进行实验。很多此类技术都先在大学的对照试验中被研究过，然后由SEL用于实际的项目中，比如逐步抽象的代码阅读法（Coding Reading Stepwise Abstraction）[11]、净室方法（Cleanroom）[21]以及面向Ada与对象设计方法（Ada and Object oriented

design）[18]。大学研究减小了SEL使用这些技术的风险。随着时间的推移，我们了解了如何组合对照试验与案例研究，以便对独立活动进行正规的分析。

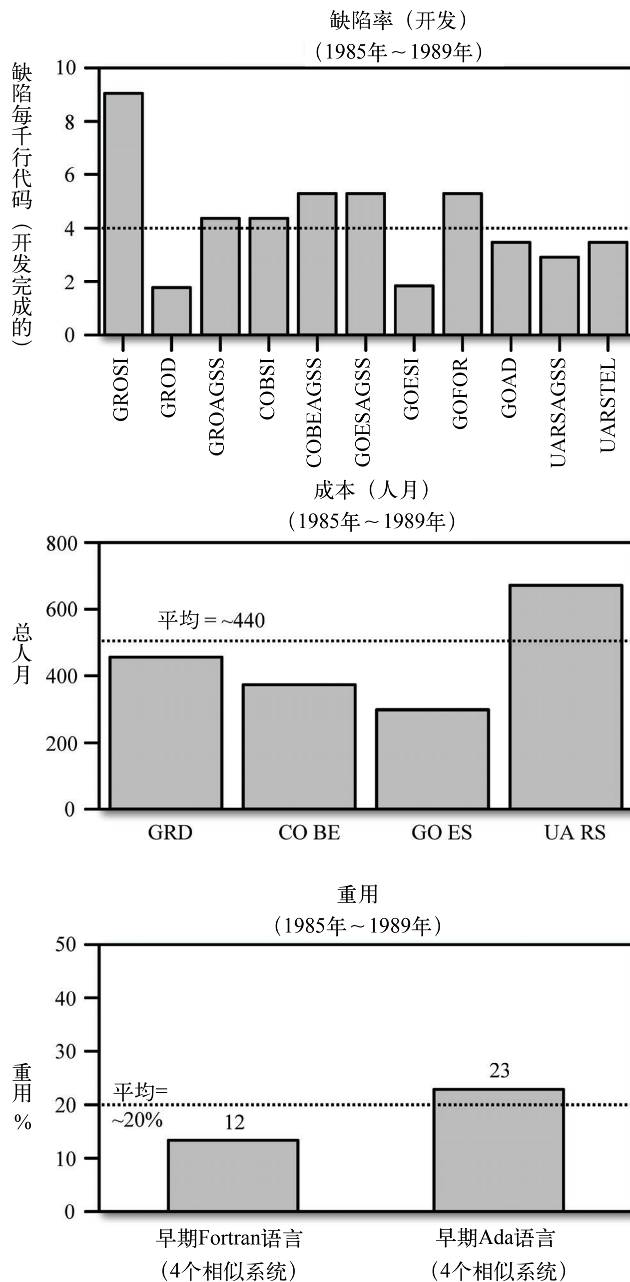


图5-2　NASA的基线示例

我们首先对新技术进行实验，以研究流程应用与相应产品特征之间的关系。这些技术的选择是基于我们在SEL中观察到的各类问题，同时以特定的目标为对象，比如最小化缺陷。为了识别需求中的问题，我们开发了一系列阅读技巧，以便从需求文档中发现缺陷[17]。

我们清楚地意识到选择正确的流程对于建立符合预期的产品特性的重要性，并且某种形式的评估与反馈对于项目控制来说很有必要。重用流程，产品以及其他形式知识的经验是改进的关键。随着经验的增长，我们也学习了适应与改进技术。

5.4.4 执行流程

执行流程开始时，采集数据成了一项额外工作。我们希望开发人员在正常进行开发流程的同时，填写我们的数据表。我们随意地监测了一下这个流程，以确保开发人员理解并填写了数据表。终端应用一致性和支持工具的缺乏迫使我们不得不手动采集数据。我们与开发人员分享中间结果，以便他们提出反馈，修正错误的理解，并且提出更好的方法采集数据。随着时间的推移，我们使用GQM手动采集的数据越来越少，我们把数据采集直接嵌入到开发流程中，这样数据更加准确，开销更少，并且使我们能够评估流程的一致性。通过开发人员与实验工作人员的互动，我们获取了开发人员经验的详细信息，为局部需求和目标提供有效反馈。然后，通过普遍的信息反馈流程合并对照试验与案例研究，对特定方法技术提供更正规的分析。

5

5.4.5 分析

为了表征环境，我们首先要建立分析基线。基线由很多变量组成，其中包括工作内容、缺陷类型，甚至是随着时间的代码增长。这些变量向我们提供关于环境的见解，启发我们应该着重改进哪些方面，并且让我们更好地理解项目之间的相同与不同之处。我们使用实验范式来分析软件开发研究，即实验的设计、评估、反馈对于学习都至关重要。分析方法的变迁从分析变量间联系[15]开始，到建立回归模型[1]和更加复杂的量化方法[19]，最后到包含各种形式的定性分析方法[20]。在学习流程中，定性分析十分重要，使我们认识到各种现象的起因。我们一步一步应用从观察与反馈中产生的想法来改进我们的理解，同时把它们融入模型当中，指导我们应该在何时何处使用更加正规的分析方法。

我们也意识到执行一个涵盖所有环境变量的大规模有效实验是不可能的。所以从预先实验的设计和准实验中得到的启发十分关键，并且我们把它们与对照试验结果、案例研究结合在一起。

5.4.6 封装

我们对于封装实验结果的微妙性、重要性与复杂度的理解发展得很缓慢。我们在一开始记录了基线与模型。然后，我们认识到目标明确的、可修改的技术包的重要性，比如通用代码组件或者可针对具体项目修改的技术。我们的实验结果需要在环境中使用，并且需要不断地根据我们的经验改变环境。技术转移包括一个新的组织构架、实验和循序渐进的文化改变。我们建立起实验模型，即在特定机构中基于观察和反馈的行为模型。我们为流程、产品、缺陷和质量建立专门的

可修改的模型，用各种方式封装我们的经验（例如方程、直方图和参数化流程定义）。难点在于合并这些封装好的经验，所以诞生了经验工厂组织（Experience Factory Organization）[3]，参见图5-3。

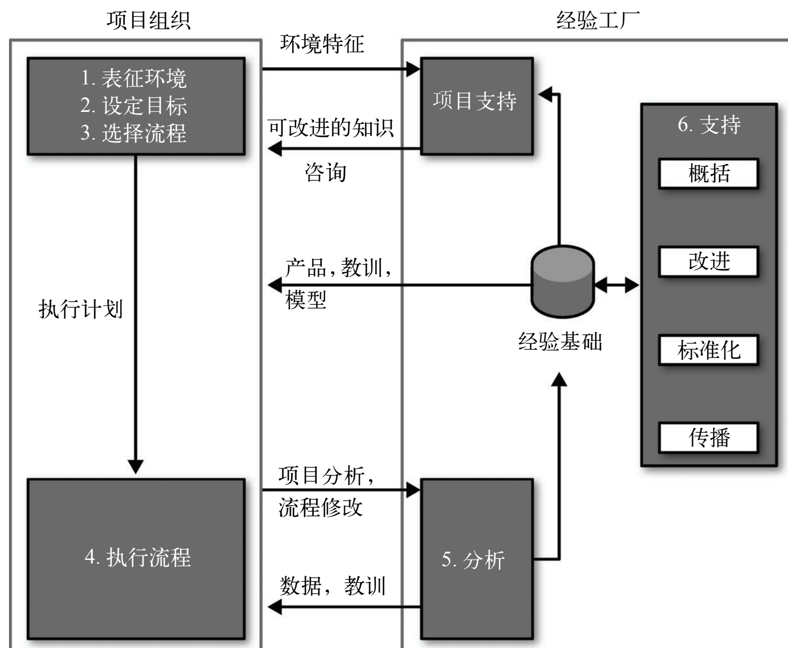


图5-3　经验工厂组织

经验工厂的流程、产品和其他形式的知识对于改进十分重要[3]。并让人认识到从组织内部取的经验和获得的知识需要和项目开发组织拆分开来。典型的项目组织活动把一个问题分成简单的几份，解决方法的实施，设计与实现，验证与确认。目标是在成本之内按时完成项目。经验工厂的活动包括统一不同的解决方案，重新定义问题，一般化、形式化和整合经验。通过综合分析观察结果，用实验测试各种意见来完成。其目标是获取经验并为项目提供推荐意见。它的主要责任是评估、修改、封装经验以备重用。

经验工厂不能单纯基于小型的、已验证的实验。人们需要观察实际应用中的意见、概念、流程等，从中得到启发，加之理性的判断，以此为基础建立经验工厂。

只有当我们知道如何从观察中学习，并且结合案例研究与对照试验的结果（实验结果比我们的看法更容易发表），我们才能继续应用、观察、学习这一流程。我们也触及到现成构件开发（Commercial Off-The-Shelf，COTS）、阅读技巧等。

在一个组织中的学习过程是耗费时间且循序渐进的，为了让我们的开发团队保持兴趣，我们需要提供一些短期的结果。我们需要找到途径加速学习的过程，得到中间结果，更快地反馈到项目中。如果我们的演进过程是成功，那么基线与环境也会一直变化，所以需要不断地重新分析环境。我们需要谨慎地在优化与复用经验之间做取舍。

5.5 结论

本章中讲述了我们在超过25年的时间里的演变并针对特定环境定制目标和流程的过程。我们在三个时间点上（1987，1991和1995）使用三个数据来衡量持续改进的效果：开发缺陷率、减少的开发成本以及代码重用的改进。每一个数据表示每个年份前三年间的平均数，参阅表5-1。

表5-1 在SEL中应用QIP的结果

SEL中的不断改进流程	1987年～1991年	1991年～1995年
开发缺陷率	75%	37%
减少的成本	55%	42%
代码重用改善	300%	8%

在这段时间内，在建系统的功能不断增加。据一个独立研究的结果显示，从1976年至1992年，功能增加了5倍。

该研究耗费了10%的开发经费，但这10%经费带来的改进却显示出一个数量级的改进，投资回报率很高。

但表5-1中的数据并不是一个对照试验甚至良好定义的案例研究的结果。即使我们什么也没有做，它们也许也会发生。因为没有对照组。但我们有不同的看法，我们相信这是学习流程的结果。在我们不知道从何时开始，将要做什么，又会如何发展的情况下，不可能对这项25年的研究进行对照试验。

在SEL的25年中，我们对软件改进有了深刻的了解。我们的学习流程也和软件开发改进本身一样，不断改进。我们把所学到的应用到流程，产品以及组织构架中。演变流程依赖于研究与实践的共生关系。这种关系需要双方面的耐心和理解，但是一旦成熟，就会带来收益。

我们在应用中学习，并适当地配合了各种预先实验设计，准实验，对照试验和案例研究，并得出了丰富的实践及理论结果，这样的结果是无法从单纯的正式试验中得到的。此外，这种大规模学习流程还促成了诸如GQM、QIP和经验工厂等多种技术和方法的评估和发展。

5.6 参考文献

[1] [Bailey and Basili 1983] Bailey, J. and V. Basili. 1983. A Meta-Model for Software Development Resource Expenditures. *Proceedings of the Fifth International Conference on Software Engineering*: 107-116.

[2] [Basili 1985] Basili, V. 1985. Quantitative Evaluation of Software Methodology. Keynote Address, *Proceedings of the First Pan Pacific Computer Conference* 1: 379-398.

[3] [Basili 1989] Basili, V. 1989. Software Development: A Paradigm for the Future. *Proceedings of COMPSAC '89*: 471-485.

[4] [Basili 1997] Basili, V. 1997. Evolving and Packaging Reading Technologies. *Journal of Systems and Software* 38(1): 3-12.

[5] [Basili and Beane 1981] Basili, V., and J. Beane. 1981. Can the Parr Curve Help with the Manpower Distribution and Resource Estimation Problems? *Journal of Systems and Software* 2(1): 59-69.

[6] [Basili and Freburger 1981] Basili, V., and K. Freburger. 1981. Programming Measurement and Estimation in the Software Engineering Laboratory. *Journal of Systems and Software* 2(1): 47-57.

[7] [Basili and Green 1994] Basili, V., and S. Green. 1994. Software Process Evolution at the SEL. *IEEE Software* 11(4): 58-66.

[8] [Basili and Hutchens 1983] Basili, V., and D. Hutchens. 1983. An Empirical Study of a Syntactic Complexity Family. *IEEE Transactions on Software Engineering* 9(6): 664-672.

[9] [Basili and Reiter 1981] Basili, V., and R. Reiter, Jr. 1981. A Controlled Experiment Quantitatively Comparing Software Development Approaches. *IEEE Transactions on Software Engineering* 7(3): 299-320 (IEEE Computer Society Outstanding Paper Award).

[10] [Basili and Rombach 1988] Basili, V., and H.D. Rombach. 1988. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering* 14(6): 758-773.

[11] [Basili and Selby 1987] Basili, V., and R. Selby. 1987. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering* 13(12): 1278-1296.

[12] [Basili and Weiss 1984] Basili, V., and D. Weiss. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* 10(3): 728-738.

[13] [Basili and Zelkowitz 1977] Basili, V., and M. Zelkowitz. 1977. The Software Engineering Laboratory: Objectives. *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*: 256-269.

[14] [Basili and Zelkowitz 1978] Basili, V., and M. Zelkowitz. 1978. Analyzing Medium-Scale Software Development. *Proceedings of the Third International Conference on Software Engineering*: 116-123.

[15] [Basili et al. 1983] Basili, V., R. Selby, and T. Phillips. 1983. Metric Analysis and Data Validation Across FORTRAN Projects. *IEEE Transactions on Software Engineering* 9(6): 652-663.

[16] [Basili et al. 1995] Basili, V., M. Zelkowitz, F. McGarry, J. Page, S. Waligora, and R. Pajerski. 1995. Special Report: SEL's Software Process-Improvement Program. *IEEE Software* 12(6): 83-87.

[17] [Basili et al. 1996] Basili, V., S. Green, O. Laitenberger, F. Shull, S. Sorumgaard, and M. Zelkowitz. 1986. The Empirical Investigation of Perspective-Based Reading. *Empirical Software Engineering: An International Journal* 1(2): 133-164.

[18] [Basili et al. 1997] Basili, V., S. Condon, K. Emam, R. Hendrick, and W. Melo. 1997. Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components. *Proceedings of the Nineteenth International Conference on Software Engineering* (*ICSE*): 282-291.

[19] [Briand et al. 1992] Briand, L., V. Basili, and W. Thomas. 1992. A Pattern Recognition Approach for Software Engineering Data Analysis. *IEEE Transactions on Software Engineering* 18(11): 931-942.

[20] [Seaman and Basili 1998] Seaman, C., and V. Basili. 1998. Communication and Organization: An Empirical Study of Discussion in Inspection Meetings. *IEEE Transactions on Software Engineering* 24(7): 559-572.

[21] [Selby et al. 1987] Selby, R., V. Basili, and T. Baker. 1987. Cleanroom Software Development: An Empirical Evaluation. *IEEE Transactions on Software Engineering* 13(9): 1027-1037.

第6章

性格、智力和专业技能对软件开发的影响

Jo E. Hannay

软件工作中最重要的因素不是程序员所使用的工具和技术，而是他们的自身素质。

——Robert Glass（Fact 1 of Facts and Fallacies of Software Engineering [2002]）

按照“个体差异”研究，优秀的程序员比普通程序员要好上28倍。考虑到他们的薪酬从来不是与之相称的，所以他们是软件领域最合算的买卖。

——Robert Glass（Fact 2 of Facts and Fallacies of Software Engineering [2002]）

Robert Glass的两个假说刊登于IEEE计算机学会网站的一篇名为“软件工程中被遗忘的基本常见行为”的文中。这两个假说反映了几个重要问题。最首要的是，他们把焦点放在计算编程中“人的问题”上而不是其他技术方面。这种聚焦帮助了实证软件工程的研究学科，使它从其他角度（管理学、社会学和心理学）来讨论软件工程实践。有人会说，为了使软件工程实践更加以证据为基础，这样做是必要的。

为便于讨论，我们假设Robert Glass的两个假说成立。请注意，不断重新评估和完善这种言论已经是我们的一项工作，即使它们可能是基于实证的。知识永远不会是静态的，因此Glass的观点可能需要做一些改进。但是，先把这个问题放一放。我们现在感兴趣的是其所引发的三个问题，它们同时也自成一体。这些问题把软件开发作为一个整体，而不只有编程，因此更具有普遍意义。

- 你能确切定义出优秀的软件开发者吗？
- 如果可以，你能否找到可靠并有效的方式确定一名开发人员比其他人更优秀？
- 如果你不能，你是否应该更专注于工具和技术？

如果能确定地回答前两个问题，其意义不亚于一场小革命，Joel Spolsky[①]这样的人会很高兴。他曾写道，聘请巨星程序员（而非普通程序员）将决定你是创造出一个能够推进世界前进的真正

① 世界最具影响力的程序员网志Joel on Software的主人，当今最热门技术网站Stack Overflow创办人。其作品《软件随想录》已由人民邮电出版社出版。——编者注

伟大的产品，还是只产出平庸的代码，使团队中的其他人陷于不断的重新设计、重新构架和调试的痛苦中[84]。然而，事实是很难找到顶尖程序员。长期从事招聘软件专业人员的人对于如何发现高端程序员有一些合理的意见和有用的建议（参见Spolsky的文章），但问题依然严峻。通常，如何寻找、留住、发展人才是人力资源经理们的首要问题，但2010年全球评估趋势报告[26]显示，依然缺乏一套以证据为基础的方法来解决这些问题。

第三个问题涉及关注个人能力（例如，技能和个性）与关注辅助环境（例如，使用工具或者结对编程）的区别。雇用最好的程序员是不够的，正如航空公司聘请最好的飞行员也不足以减少空难发生的风险；还必须在环境中加入许多安全特性，并且需要与人合作的能力。

在人的专业技能与工具之间谋求最佳平衡并非易事。但如果不能定义专家，那么你可能没有机会来平衡任何东西。此外，如果你自认为知道怎样的人是优秀的程序员，但你的直觉却是错误的话，那么你的项目处境可能会很悲惨，除非有环境措施能够发现问题，并能对不可预见的紧急情况进行调整。于是Glass观点的可操作性依赖于能够可靠地从人群中找出聪明人。正如我们将要看到的，这不容易做到。软件工程涵盖各种任务，当前任务的性质影响了人们对专业技能的判断能力。

让我们从编程开始说起。

6.1　如何辨别优秀的程序员

我们需要考虑两个更深层次的问题来确定是否能够定义优秀的程序员。

- 程序员身上的哪些因素使其有优秀的编程表现？是否因为她的经验？或者是她的个性？又是否需要测量她的智商？结对编程和团队工作时，这些问题的答案又如何？
- 什么是优秀的编程表现？例如，你应该雇用单位时间写代码最多的人还是不管所耗时间，代码质量最高的人（无论质量是如何定义的）？

这些问题触及了最基础和最困难的部分。工作中，当人们需要做出战略决策时，这些问题都或明或暗地由直觉决定，或者根据多少有些站不住脚的个人档案和能力测试决定。在学术研究中，这些问题有着更深入和更广泛的意义，于是催生出大量跨越学科的研究工作，试图深入理解这些问题。

6.1.1　个体差异：固定的还是可塑造的

把人与人区别开来的各种特征，研究人员简称之为*个体差异*，可以在从*固定*到*可塑*之间的连续区间上归类。在固定的一端，你可以找到个性和认知倾向这类在人的一生中被认为相对稳定的特征。在可塑的一端，则包括任务相关技能、知识和动机等这些被认为受短期环境、培训和学习影响的东西，因此可以受到人为操纵，比如说是可以改善的。

由固定特征来区分人是一种比较快捷、便宜、简单的方法；评估和发展人的技能却是一个比较漫长的过程。也许人类的天性就是根据一些模式来评价人，这就难怪，招聘行业会有一系列的测试来度量一个人的固定特征。

对于如何评价一个人，我们有着各种各样的意见。基于固定特征的招聘是不是会涉及歧视？让工作人员参加智力或性格测试会不会不道德？在一些国家（特别是在欧洲），这样做的确会被

视为不恰当，并且众所周知，智商测试是带有种族偏见的（在美国这样做会被起诉）。但是各个行业和政府仍然以各种名目继续他们的测试。那么，工作绩效到底是由固定特征，比如个性、智力，还是由可塑特征，比如技能和专业，决定的呢？

6.1.2　个性

个性已经在编程和软件工程方面受关注一段时间了。例如，Weinberg在《程序开发心理学》① 一书中预测到“重视个性将对提高程序员的表现有很大的贡献”[90]，并在1998年版的该书中重申了这一观点[91]。Shneiderman在《软件心理学》（*Software Psychology*）中指出：“个性因素决定了程序员之间的互动以及每个程序员的工作方式。”[79]不过，两位作者都承认在个性对个人表现的影响上缺乏实验证据，“依靠个性测试来判断谁将会成为优秀的程序员并不很成功”[90] [91]，“不幸的是我们对于个性因素的影响也知之甚少”[79]。

从那时起，人们已经开始进行针对个性和软件开发的实证研究。例如，Dick和Zarnett得出这样的结论：“挑选具有有利于结对编程必要个性特点的人组成的开发团队，会比根据纯技术能力组建的团队，在极限编程上获得更大成功。”[22]关于具体编程任务之外的事宜，Devito Da Cunha和Greathead总结道：“如果一家公司根据员工的个性类型和潜能组织他们，工作效率和质量可能会得到改善。”[21]并且有人认为应该把各种个性映射到软件开发的特定角色上[15] [2]。

那么，什么是个性？平时我们总是能谈到个性。“她有着成为律师的完美个性”或“他的个性真的有问题”。的确，我们每天都基于这样非正式的直觉做出重要的决定。也许其中一些后来被证明是正确的，有些则是错误的。采用循证方法意味着用系统化手段（读取的，科学的）得到的信息做出决定。需要说明的是，这种系统化的做法有其内在的局限性，关于这点之后会有更多的讨论。

首先必须明确，是否有可能定义和度量个性。这应该是可能的，但也许对我们这样的技术人员不太明显。我们或许很快就能找到切实方法来识别个性（比如，遗传的或心理生理的）[69]。然而，个性理论近百年来的主要发展一直遵循着另一条路径。科学家们推断，我们所谓的基于人们日常行为和言论的“个性”之间存在差异。基于所述的行为和语言推论的研究路线已经建立了一些令人信服的人格模型。其中最知名的也许是“大五模型（Big Five Model）”，以及相关的“五因素模型（Five Factor Model）”（参见以下的“个性因素”）。所以，是的，个性是可以定义并且度量的。讨论一个人的个性是有科学意义的，并能够依靠测试确定一个人的个性。科学家们认为，我们已经构建了有效的个性概念。

个性因素

个性模型有很多种，并且每种模型都有自己的测试来度量个性。个性测试被广泛用于商业和政府活动，比如招聘、职业咨询机构和军队。虽然其中的某几个测试源于心理学研究的理论和实证基础，但是它们中的很大一部分随着时间的推移出于某些目的而被

---

① *The Psychology of Computer Programming*，作者为Gerald Winberg，中文版由清华大学出版社出版。——译者注

简化或改变，已经很少或几乎没有什么科学依据。（个性测试的历史和重要事件请参照Paul的报告[66]。）

与此同时，学术界中的个性研究建立起了经过充分研究的模型和测试。近年来，有两个模型起到了主导作用[6]，它们由五个因素组成，被叫做五因素模型（Five Factor Model，即FFM）[19]和大五模型（the Big Five）[32] [33]。FFM认为个性是处于一个遗传因素、环境影响的综合模型中。大五模型认为人们生活中最重要的个性差异会在他们的自然语言中体现出来，所谓的词汇假说（Lexical Hypothesis）[32]。这两种模型往往被视为一体，他们的相关因素协调地相当不错，例如参见Goldberg 等人的研究[34]。然而，这两个模型在概念上是不同的，理论依据的不同使得设计测试这些因素的方法也不同。

这5个因素如下。（描述摘自Pervin和John的报告[68]。）

(1) 外向性

评估人际交往的数量和深度，活跃程度，刺激需求，喜悦的能力。

(2) 亲和性

评估人际关系取向的特质，思想、情感与行动上从同情到对立的程度。

(3) 尽责性

评估个人在目标指引下行为的组织度、持久度和动机。将可靠、高标准严要求的人与懒散、马虎的人作对比。

(4) 情绪稳定性/神经质

评估情绪调节与情绪稳定。识别容易产生心理压力、不切实际的想法、过度渴望或冲动和反应不适的人群。

(5) 经验开放性

评估为了自我而寻求体会经验的程度，对不熟悉事物的忍耐度和探索性。

几种商用模型和测试在学术界因为理论基础差，可靠性低，有效性低而受到批评[28] [71] [72]。特别是许多性格测试有“Forer效应”①，他们之所以引起所有人共鸣是因为其描述笼统、含糊，而与真正的个性并没有关系。

人们通过多种方式用个性的定义对人进行分类。例如，按照大五模型，我们发现程序员属于同一个参考群体，即外向性较低，情绪稳定性较低，经验开放度高[41]的群体。相关结果另见Moore[65]、Smith[80]、Woodruff [32]、Capretz [14]、Turley 和Bieman[87]的报告。程序员们作为一个整体也更加趋同，也就是说，他们的性格比较相近。这证实了对于程序员的刻板印象是神经质、内向、聪明，顺便说一下，还有男性（我知道一些人认为这等同于一个性格特征）。

正如我刚才所说，系统的科学方法除了优势之外也存在固有的局限性，而这种局限也部分来

① B.T.Forer[27]对他的学生做了一个个性测试。然后，他丢弃了学生的反馈，从占星术的书上复制了完全一样的个性分析给所有学生，并且要求学生们用从低到高五分来评价各自个性分析的准确度。评估结果的平均数4.26。Forer重复该实验无数次，平均数大约保持在4[23] [42]。该Forer效应也被称为“巴纳姆效应”[62]。

源于它的优势。其中一个限制就是科学家们所称的内容效度：如何合理地证明个性定义涵盖了“所谓”个性的全部？例如，大多数人不认为个性只有五个因素。其他模式尝试捕捉更复杂的类型，并创造出了微妙、复杂的人格因素组合，但这些模型缺乏建构效度。系统化本身就限制了在哪些方面能够系统化！这里我们遇到了典型的权衡问题。简单的模型（如五因素）简化了现实，能经得起验证和测量。更复杂的模型看似能更好地把握现实，但难以验证。

对我们来说，最终的问题是个性测试的效用。个性上的一点差异会造成多少表现上的差异呢？利用一个人的个性来预测其他行为，比如说编程效率，有多大的可行性呢？已经有大量研究分析了个性和一般工作绩效之间的关系。研究结果已经在一次大型的元分析中被总结出来[6] [67] [7]。根据Barrick等人的研究，个性对于工作绩效的影响是“有些令人失望”和“即使在最好的案例中……也是一般的”[6]。因此，个性对工作绩效的直接影响可能很少。

不过，个性的社交因素可能会通过影响团队合作而对工作表现间接产生重大影响。事实上，大五因素对团队合作的影响比对总体工作表现的影响大[6]。这表明，研究个性对于协作的影响可能比研究其对于个人表现的影响更为相关。我们通过198名专业程序员在一天中结对编程的行为表现来研究这个方面[41]。结果发现，个性与结对编程效率的关系较弱。即使是粗略度量的专业度、任务复杂度、甚至程序员在哪个国家受雇也比个性的预测能力强。这项研究还分析了个性对独立编程的影响，以及结对编程效率是否会受个性影响等。专业度和任务复杂度再次强于个性的预测能力。

顺便说一下，一件事物（本文中的“个性”）对另外一件事物有多大的预测能力，被称为效标效度。有关建构、内容和效标效度请参照“实证科学的三大挑战”。

6

实证科学的三大挑战

假设你要在厨房里现有的两个橱柜之间装一个新的橱柜。你用卷尺测量了一下安放的空间。但是，你注意到卷尺末端的金属钳看起来安装得有点马虎。因为空间很有限，所以你必须要确保可用空间的大小。所以，以防万一，你又使用了折尺（你发现尺子的两端也有不正确安装的金属夹）。两个测量结果偏离了约一毫米（在昏暗中，弯着脖子时看到的极限值）。邻居给了你一个具有数字显示的激光测量设备，但它的精度只有半毫米。似乎你永远也不会十分清楚到底有多大的空间，但你决定购买橱柜，即使在最糟糕的情况下，你觉得橱柜多少可以挤一挤吧。

这个场景中你的问题在于测量的工具，它们都不够精确。为了解决这个问题，你用了几种测量仪器增加信心，至少让你知道不会差得太远。你也许感觉很沮丧，不过与测量非物质概念相比，比如说心理特质与程序员的水平，测量物理对象（地球上的宏观物质）并不那么困难。与物理对象不同，非物质的概念首先很难定义。（在某种意义上，物理对象也存在这个问题。我们并不真正知道物理对象由什么组成和它们之间的空间，不过使用我们对于“物理对象”的通常理解在一般情况下不会有什么问题。）

实证科学面临的第一大挑战就是建构效度（construct validity）：你怎么知道你使用的测量仪器测量了你想测量的东西，或者说首先你真的知道你要测量的是什么东西吗？一个概念与其测量手段（其指标）一并被称为建构，参见图6-1a。例如，大五人格因素

模型的每一个因素是由20个问卷问题度量得出，因此“外向性”的建构表示为一个标有“外向性”的椭圆形，然后有20个指标框指向它。

通常情况下，建构效度是通过自上而下并自下而上地攻破问题而达成的：一个人设计出自认为能代表所度量概念的度量方法，这么做的同时，他也对这个概念有了更充分的了解，然后指引他设计进一步的度量方法，如此自举式反复。任何测量方法都需要遵守一定的统计标准，这些标准被执行的程度保证了建构的质量和意义。实现高建构效度是很困难的，而且通常需要多年的研究才能使它有实用价值。商业应用通常没有时间这么做或者根本不关心，因此，世界上那么多的测试事实上可能什么都没有测量到！

第二大挑战是内容效度（content validity）。一个建构代表了一个概念中有科学对照的部分。然而，重要的是要记住，建构可能不仅仅代表一个概念。内容效度，表示了一个建构在何种程度上合理捕获了我们对一个概念的理解，见图6-1b，其中一个建构被视为只覆盖了一个概念的一小部分。人们需要不停地努力以确定是否应该扩大概念中受科学对照的区域（图6-1b中的虚线椭圆形）。科学最傲慢的地方之一，就是声称它所控制的就是一切，如智商测试测量了人类的全部智慧。

第三大挑战是效标效度（criterion validity）。这涉及一项建构能否用于预测其他建构中的变化。你可以使用个性预测编程效率吗？可以用编程技巧来预测一名开发人员在下一个大型开发项目中的效率吗？如此种种。见图6-1c，一个建构中的变化（即预测）预示着第二个建构中的变化（效标）。

这里人们很容易犯错，混淆建构效度和内容效度。例如，如果你想定义一个合作的“建构”，然后想用这个建构来预测球队表现的建构。很容易简单地把合作定义为会带来好或坏的表现的建构，也就是说，好的合作是一切带来出色成绩的合作。这就自动给出了好的效标效度，但实际上，合作建构本身并不是一个真实的建构。相反，它只是团队表现的一个方面。

个性影响也很有可能在一段时间之后才表现出来。如果结对编程只进行一天，双方的个性影响可能不会十分明显。几乎能自动进行的短时间测验（如个性与智力测验）与需要专家分析的整体评估方法之间，也存在争论。以一个“渴望成就”的人格特质为例。元分析认为，有争议的主题统觉测验在被用于测量“渴望成就”特质时，实际预测的有效性比标准问卷更好。而以标准问卷为基础的测试，在受控环境中的预测性能较好[82]。①TAT继续受到科学界的怀疑。然而，Spangler说：“科学方法可能会不经意地最小化了个体表达差异、个体互动差异和环境特性。”[82]这里你的内容效度再次受到挑战！还要注意的是，度量实际生活中的长期表现比度量实验室可控条件下的表现要难得多。

① 这里还有其他因素：TAT的效标效度也依赖于任务活动有关的刺激鼓励，而问卷的效标效度是依赖于与社会成就相关的刺激鼓励。这表明，研究成果没有想象的那么独立，但它们仍然是值得考虑的。

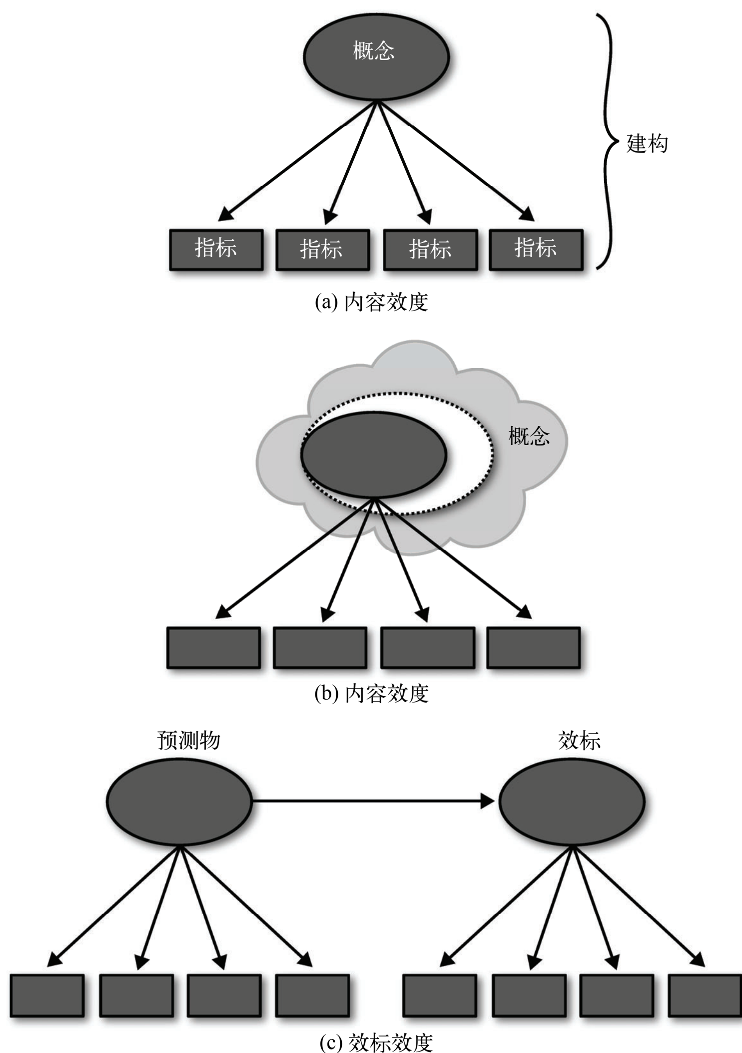


图6-1　有效性类型

总之，个性似乎不是一个非常强大的效率预报器。让我们简要地看看有哪些影响因素。一般情况下，预测学术和工作表现时，“责任心”是最持久稳固的个性因素[74]。然而结对编程中恰恰并不如此[73]。事实上，我们在数据中甚至发现，责任心有负面效果（也适用于独立的程序员）。看似有积极作用的是经验开放性[73]和不同水平的外向性[41]。不用外向性水平的结对比水平相似的结对工作得更快。

6.1.3　智力

有一位大型软件开发公司的人事经理曾表示，如果有机会试用一种科学验证过的测试来度量

编程能力的话，预测编程效率有且仅有一个因素，即智商。这种观点很有意思，也相当说明问题：在某种意义上，它并没有错，但它太过离谱，会引你走上一条错误的道路。

一般心理能力（General Mental Ability，GMA）是一个智力或认知上非常笼统的概念，被认为存在于每个人。这就是智商测试大体上测量的东西。更多关于“智力”的内容请参阅下面的“智力因素”。GMA是一个相当好的预测学习能力的指标[74]。换言之，GMA高的人会比GMA低的人更快地学习新任务。GMA也可以相当不错地预测没有经验的员工未来的工作表现[74]。因此，如果我们的人力资源经理想要雇用没有任何大型复杂系统工作经验的程序员时，她也许没有错，的确应该考察他们的智商。

GMA的效果依赖于手头任务的类型。如果一直以来，某个任务的最佳表现者总是用相似的策略来解决它，那我们认为这个任务是稳定的。而不稳定的任务，则是用不同的方法来解决的。当人们在稳定的任务上获得了技能之后，智力水平的影响在经过一段时间后会减小，很快工作绩效会依赖于其他因素，如经验和培训[1] [76] [75]。高度依赖于相关领域知识的任务，比如软件开发等，就是如此[10] [1]。软件开发总的来说，特别是编程，同时带有稳定和不稳定的任务。这意味着，智力固然重要，但不是全部。对于有经验的工作者来说，GMA本身并不是终极预报器[25] [74] [1]。那么究竟什么是终极预报器呢？答案是：智力和技能[74]。

这些研究告诉我们单单依靠GMA长期来说是不明智的。GMA的问题在于它过于笼统。它被认为是普遍存在的人类特征，而测量手段被故意设计成与任何任务领域无关。（依赖GMA的缺点也包括，人们观察到GMA并不那么普遍，它也依赖于其他东西，比如说文化[11]。）这就是说，如果你想要雇用优秀的软件开发人员，除了需要评估普遍能力之外，也需要评估一些与工作相关的东西。

智力因素

现代科学观点中的智力包括几个方面。一个被许多研究人员接受的智力模型是，卡洛尔模型[16]，它包括以下8个主要因素。

Gf：流体推理

Gc：涵化知识

SAR：短期理解与短期工作记忆（STWM）

TSR：长期记忆

Gv：视觉处理

Ga：听觉处理

Gs：加工速度

Gq：数理知识

这些因素（每一个都有一些子因素）包括被怀疑有与文化相关的部分（如Gc，涵化知识，个人吸收主流文化知识和语言的程度，这就是为什么智商测试没有公平对待少数族裔的原因），和独立于文化之外的东西（如Gf，流体推理，它代表在短期内的推理能力）。

很少有实验证据能证明一般智力（GI）中存在一个包罗万象的因素，即使几个研究

人员已经提出了一个[45]。此前，Spearman认为[83]，他的“智力概念”（简称g）代表了智力的本质，但这个概念现在已被视为Gf因素的一个对应。我们本章所说的“一般心理能力”（GMA）实际上是Gf和Gc的混合物[88]，参见图6-2。这些因素之间有着依赖关系。例如Gf辅助Gc的建立。还有证据表明，一些因素会随着年龄的增长而退步。

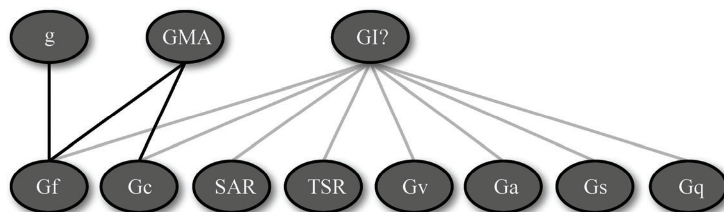


图6-2 智力因素

6.1.4 编程任务

人们常用工作样本测试（work-sample test）来测量技能。这些测试由小的，具有代表性的任务组成。该定义提出了两项挑战：“小”和“代表性”。如果你想测试潜在员工或在职员工的编程技能，你想要用尽可能少的时间。因此，测试任务需要比在正常工作中遇到的任务小很多。但是，你怎么能确定测试内容的相关度？这就是代表性的问题了。

事实上，在这里你再次遇到建构效度和内容效度：你怎么知道你的测量手段测量了你想要的东西，首先你真的知道你要测试什么吗？讨论“个性”时我们提过这样的问题，对于“智力”我们也问了同样的问题（虽然我们并没有对此做文章），在“编程技能”上还是有这样的问题：定义“编程技能”（这包括定义编程任务），然后制定一个快速测试。一般来说，工作样本测试的建构效度是有问题的[13]。

我们正在研究实验室里开发一种测量手段以评估编程技能[8]。它基于工作样本测试。工作样本测试由小型到中型的编程任务组成，随着对编程技能概念的不断了解，我们会精心选定、替换和修改这些测试。

该测量手段主要基于测量理论并符合严格的统计标准。它也基于技能和专业知识的理论，这些都是为了保证建构效度。因此，该手段有双重保障：它一方面跟踪任务难度，一方面跟踪完成任务者的技能。与智力和个性相似，个人评估结果是相对于其他参与测试的人群的。与智力和个性不同的是，编程技能与雇用和留住优秀的程序员直接相关。

6.1.5 编程表现

那么，什么是出色的编程表现？显然，代码质量是一个度量标准。“高质量”代码的定义一

直是讨论的焦点。但对于工作样本测试，有一些明显的循证度量标准，如功能的正确性，继承深度等。另一个度量标准是编写代码所花费的时间。编写高质量代码花的时间越少越好。你可能认为这两个标准在相互冲突，人们需要更多时间来编写更高质量的代码。这未必是真的[4] [9]。对于某些编程任务，一名程序员不是“行”就是“不行”，如果行，他就能很快完成它。对于其他的编程任务，花更多的时间能得到更好的解决办法。重要的是要知道任务是属于哪一种类型的。

6.1.6　专业技能

技能是可以改进的。在专业技能和学习领域 什么是真正的技能以及如何改善它，有着十分广泛的研究。专业技能一般说来（而特殊技能只有在特殊情况下）关系到一个特定任务或一组任务。这与性格和智力明显不同。

除了技能，专业技能还包括其他方面。专家知识（在此是编程知识）是专业技能重要的组成部分。经验（在此指编程经验）也是如此，请参见后面的 “专业技能的构成要素”。这些要素之间有依赖关系。例如，经验促成知识，知识促成技能。

专业技能的构成要素

专业技能包括几个方面的内容[25]。

(1) 持续经验

(2) 卓越的知识表现和组织，具体分为[45]：

- 专家知识；
- 专家推理；
- 专家记忆。

(3) 在典型任务上的可靠的出色表现（所谓的专家表现）

专业技能通常与给定领域中的特定任务相关。在某项任务上是专家并不意味在其他任务上也是专家。

度量专业技能的各方面时必须十分谨慎。例如，对于持续经验（第一个方面），往往相当简单地用工作年数来度量，但相关度更高的度量方法是更多地考虑与特定任务相关的经验[78] [81]。卓越的知识表现和组织（第二个方面）更偏重专业技能中的认知方面。人们也常用工作年数来度量这个复杂的概念，因为度量一个人的认知结构并不容易。人们也假定优秀的心理表征（mental representation）是随着相关工作经验的积累而发展起来的。这种情况比较常见，但并不一定如此。当具体谈到技能时，通常指在具有代表性的任务（即工作样本）上的优秀表现（第三个方面）。

专业技能中另外两个被广泛认可的方面如下。

- “稳定”任务的专业技能是渐近式增强的。存在一个任何人都不可能达到的最优表现。对于“不稳定”的任务，这一点就不好度量了。
- 专业出自实践。如果停止实践，你的专业技能将有可能下降到你的最佳水平之下。

智力能帮助获得技能，但它并不能决定技能。根据Cattell的投资理论，在需要深入洞察复杂关系的学习阶段，“投资”Gf（所谓的流体推理，或短期解决问题的能力，见“智力因素”）有利于获取知识和技能。最近有证据表明，这个理论也适用于编程[10]，参见图6-3。在这里，工作记忆容量（WMC）替代了Gf。WMC对编程知识具有巨大的正面影响，从而积极地影响了编程技能。然而，WMC对编程技能的直接影响似乎是微不足道的！换句话说，WMC并不会直接提高一个人的技能，它只是促进了知识的获得。同样，编程经验间接（而不是直接）影响了编程技巧，但没有WMC影响大。

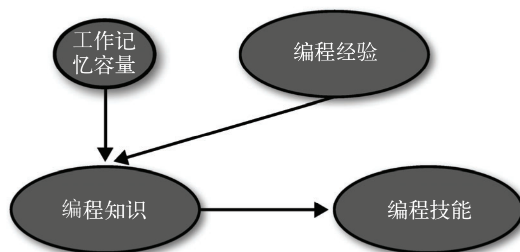


图6-3 编程技能的投资理论[Bergersen and Gustafsson 2010]

这一切证明了，对于获取技能来说，智力是一笔财富。如果你希望找到优秀的程序员，而且必须要测试他们的智力，那就测试吧，别犹豫；但是，可别忘了，最重要的是首先一定要测试他们的编程技能。

成功既需要智力也离不开技能。这样把智力与专业技能放在一起来看待能够把注意力吸引到好的方面，让人更加关注问题本身，而污辱性也更小。事实上，人们正努力把智力与专业技能合并为一个统一的理论，见下面的“合并智力和专业技能”。这里的共同点是在大脑中的认知结构。它们有些是稳定的，有些随着年龄而退化，有些通过有意的练习能得到改进。

合并智力和专业技能

认知结构是智力和专业技能的基础。人们已经努力将这两个认知主题合并为一个专业技能与智力的交叉理论[45]。有趣的对比随之产生：Gf（流体推理）是从第一原理（first principles）归纳得出的（例如，国际象棋的规则），而专家论证是演绎推理的（例如，从遇到过的和学习到的成套象棋位置中推理的）。SAR是从短期工作记忆得到，它拥有七（加减二）块信息，而专家似乎科能够利用其专业领域的专家工作记忆，这比SAR的短期工作记忆多得多。Gs是指回忆没有价值的任务的速度，专家认知速度是指回忆特定领域事物的速度。因此，人们在已有8个方面的智力中加入3个新的方面，然后获得一个组合理论。

ExpDR：专家演绎推理。

ExpSAR：短期理解和专家工作记忆检索（ExpWM）。

ExpCS：专家认知速度。

6.1.7　软件工作量估算

那么，其他软件工程任务呢？它们能习得吗？习得任务后所产生的技能可以被度量吗？让我们来看看一个出了名的棘手任务：软件工作量估算，就是估算软件开发工作量的任务。

软件系统开发一直是一个复杂的过程。估算一个大型软件开发项目工作量的复杂度更高。该任务的困难度所产生的明显影响是：软件开发的工作量估算不准确，通常都偏低很多[63]；软件开发专业人士对他们的估算总显得信心过高[56]；估算通常是不可靠的，同一个人在不同场合对相同任务会有不同的判断[35]。在过去几十年来中，估算精度似乎没有实质性的改善（以史为鉴），从结果反馈中学习似乎也很困难[36]。

除了复杂性之外，我们知道，人们在预测一项估计中的判断过程，受制于一系列无意识的过程[57] [60] [54] [50] [55]。例如，只要事先告诉人们各种基础估算，估算就很容易被操纵（即锚固效应）。人们的判断过程也对估算时所得到的信息本质和信息格式很敏感（如需求文档）[51] [52] [53] [49]。例如，问一个人需要多少时间来完成一定量的工作，与问他在一个给定的时间内可以完成多少工作，有很大的区别。后者是敏捷开发中时间盒（time boxing）的做法。结果呢？使用时间盒会增加低估的情况。此外，时间盒似乎扭转了高估小任务工作量却低估大任务工作量的倾向[38]。

与编程任务不同，并不是估算软件开发工作量次数越多，就会做得越好。有证据表明，对估算精度的在职反馈（无论是被动的、历史数据形式的，或是主动的、直接管理评估形式的）并不能改善过于乐观、过于自信或估计不可靠的问题。根据经典学习理论，这表明需要更精确的反馈和更积极的技能培训（所谓的刻意练习）。换言之，需要有意识、有目的地训练估算技能。

但问题是：这可能吗？针对性的培训要求知道针对的目标是什么。换言之，一个人需要知道什么是估算的专业技能。但是，当涉及软件项目工作量估算时，专业技能的本质似乎在躲避着我们，它不容易被观察到，因为有经验的项目经理并没有比没有经验的估算者好多少，并且专业技能的理论基础没有明确告诉我们这里的专业技能到底是什么。此外，软件工作量估计属于所谓的“定义不明”的任务。它们的成功策略甚至都难以定义，这一点仅比“不稳定”任务强一些。无论是软件工作量估算这项任务本身，或是做好这项任务所需的专业技能，都不在科学掌控之中；换句话说，我们仍在两个概念的建构效度中挣扎。

6.2　环境因素还是个人因素

现在我们转到本章开始时的第三个问题。我们现在可以把它表达得更精确一点了。如果知道了不容易度量软件开发人员的专业技能的话，你应该专注于工具和技术吗？想象一个政策制定者，她需要负责减少道路碰撞事故的数量和严重程度。道路事故是导致全世界10岁至19岁儿童死亡的主要原因，在美国可预防的死亡原因中排名第六。她应该优先考虑提高司机的驾驶技术和意识，还是应该花更多的钱建立环境保障措施，如改善道路标准，降低车速限制，或者争取在汽车上增加更安全的功能呢？你可能说这两种都需要做，但在有限的预算下呢，每项要做多少呢？

道路交通碰撞事故的研究还算处于一个幸运的位置：全球有巨大的数据可供分析。因此，合

理地决定在哪里投入资源是可能的。软件工程还没到可以做出这样明确决定的位置上。我们没有足够的数据，而且我们的任务非常多样。

6.2.1 软件工程中应该提高技能还是提高安全保障

编程能力正变得可度量。这意味着我们能更好地领会编程任务本身以及专家程序员意味着什么。我们在建构效度上已经取得了进展（也就是我们的度量方法一直能反映编程技术和任务的难度），而且也正在提高内容效度和效标效度（我们正致力于提高对编程技能和任务难度概念的科学把握，并通过实际编程中的成功来验证建构）。这使得建立和改善程序员的培训计划变得可能。

软件工作量估算却并非如此。预测一个团队或者一个项目需要多大的工作量来开发系统的一些部分，总的来说还不在我们所理解的范围内，也没有到达可以可靠地度量任务难度和估算技术的程度，这也意味着我们还不知道如何培训人们提高这方面的表现。

但我们知道，改善环境是有帮助的。这里有几个例子：从需求文件中删除无关内容，不让不确定的基础估算影响思考；估算最可能的工作量之前先估算理想工作量[48]。这些措施都是为了改变判断（即估算）发生的环境，其目的是为了减少人们在做判断时已知的心理偏见。其他环境措施包括：团体估算，一般比单个估算更准确；使用适当的过程模型，迭代开发的估算往往比接力式开发好[64]。开发支持所有这些环境措施的工具和技术是可能的。

结对编程（在Laurie Williams撰写的第17章中有讨论，Jason Cohen撰写的18章也有一小部分相关内容）就是采用环境措施改善代码生产效率的一个例子。人们依靠社交过程而不是程序员自己来提高单个程序员和团队的工作效率。重要的是要知道在何种情况下结对是有益的。例如，结对似乎对新手参与复杂的编程任务最有益[5]，也可以参阅Hannay等人[40]或Dybå等人[24]关于元分析方面的内容。

6.2.2 合作

团队合作和协作现在非常流行。例如，人们通常认为利益相关各方参与软件开发过程是有益的。因此，引入更明确的合作关系是一个主要的环境推动力。

应该值得看一下究竟什么是成功的合作。因此，我们应该观察一下人们是如何合作的，而不是（或者说除了）观察个性。人们对于合作、团队组成、群体过程（group process）有着诸多研究。事实上，几乎是太多了！我曾试图找出与结对编程有关的基本群体过程（见图6-4），但我很快意识到，除非知道合作是什么样的，否则就不可能决定运用那种理论。相当一部分研究人员经常会事后为他们的观察生搬硬套一些理论[39]。这样做很容易所以也经常发生。因为总是有一些理论是适合的！例如，社交促进或许可以解释为什么团队表现良好，而社交抑制或许可以解释为什么一支团队表现得不好。所以，你可以使用这两种理论中的任何一种来解释任何可能观察到的结果。当然，只有当你真的不知道是怎么回事时，这才可能。因为如果你知道，你应该就会看到，至少一个理论的背后运作机制与事实不相符。

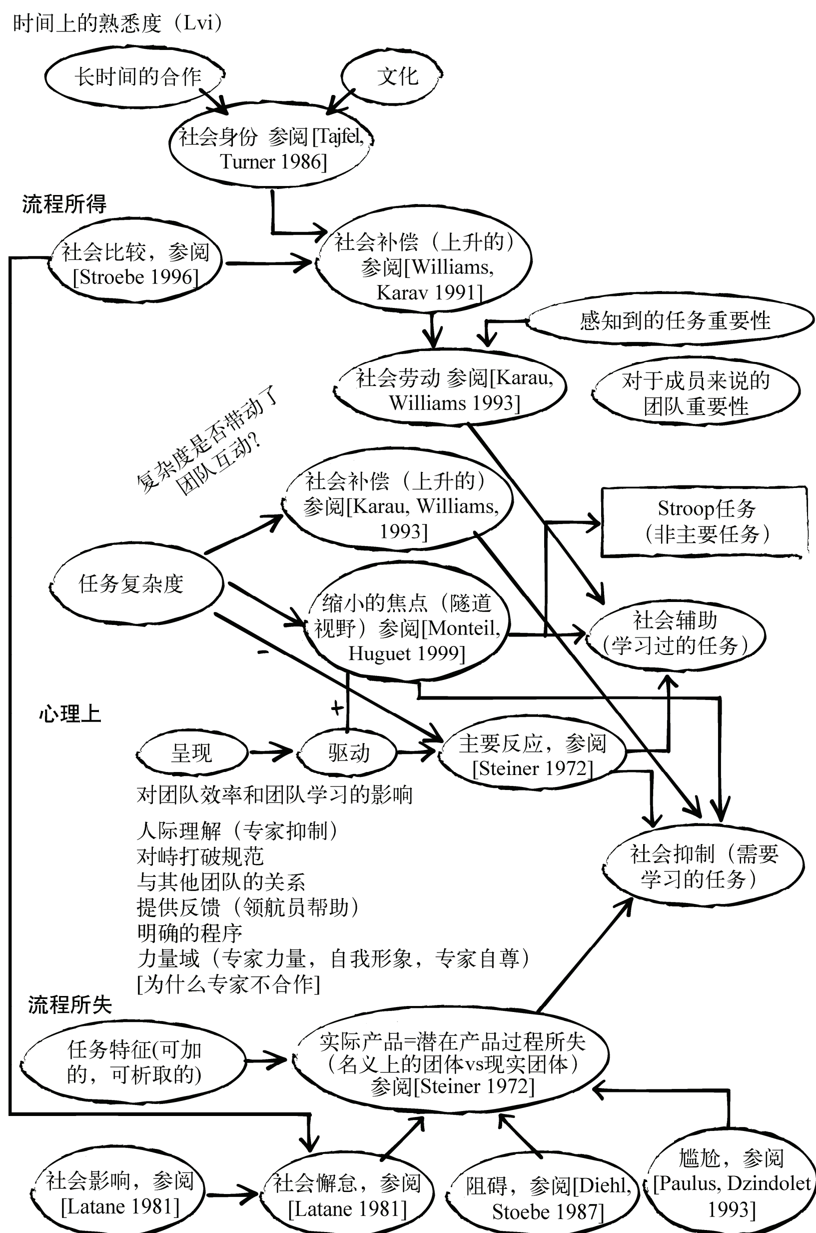


图6-4　理论太多了（如果你不知道到底怎么回事的话）

为了找出结对编程中的合作是如何进行的，我们听了43对专业程序员在解决一个结对编程问题时的录音带。合作方式根据他们的对话进行分类[89]。我们所用的分类策略是用自举的方式开发的，既从现有的策略开始自上而下构建，也根据录音当中明显而具体的合作方式自下而上搭建。你可以在图6-5中看到该结构。

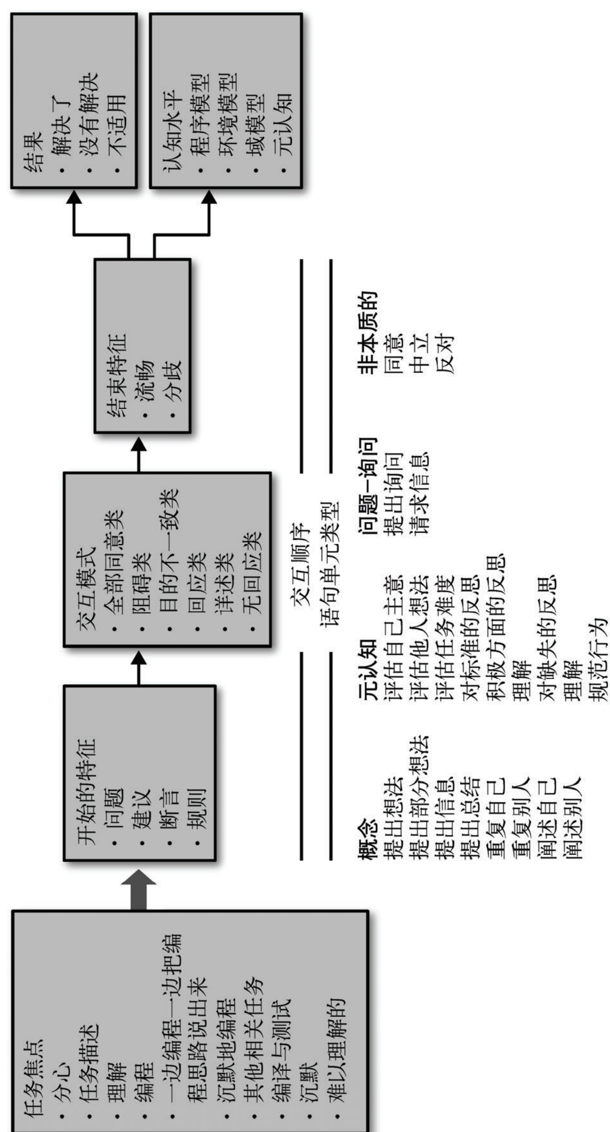


图6-5　口头合作的分类结构

该构架有两个层次。第一个层次主要是对话的焦点。例如，结对程序员是否在讨论任务的描述，或试图理解代码，或在一起写代码，或谈论别的东西，比如昨晚的足球比赛。第二个层次分析了交互顺序[44]。这里的核心是交互模式。例如，如果两个人在对话中都贡献了实质性言论，而且发言者在前者的基础上做出了进一步的发言，或是澄清了前者的发言，那么这样的对话被归类为“详述类”。

人们会认为“详述类”对话的存在是有益于工作效率的。不过，我们没有在数据中找到相应证据。目前的结论是花更多的时间来讨论任务描述会减少解决整个任务所花的时间。这时得出最终结论还为时过早，所以你不应该把这个作为证据，而应把它加入实践中的反思过程。如何理解软件工作量预估中的合作还在进一步的探索中[12]。

6.2.3 再谈个性

提到合作，不得不再次提到个性。例如一些人类学研究关注个性问题与软件工程团队合作中的意见分歧。他们发现分歧很糟糕，但是如果没有争论的话（轻度的分歧）结果会更加糟糕[58][59]。有人争论说，结对或者团队中的成员个性过于相似会导致缺乏争论。这在Williams等人[92]和Walle与Hannay[89]的报告中被实验证实了。特别是，外向性的不同产生的影响最大：结对中的俩人如果外向性的指数不同，那么他们的合作会更深入（就是有更多的讨论），胜于那些外向性水平相似的结对。

6.2.4 从更广的角度看待智力

智商（IQ）高的人不一定擅长于计划或者制定优先级，比如什么时候应该计划，什么时候应该前进并采取行动[86]。计划很重要。我们的初步调查结果显示，最快的结对程序员花了相对更多的时间来了解任务的描述，这可以看成是在计划上花了更多时间。有人进一步认为，“经典”智力测验的重点是处理速度，只是表明高分数更擅长解答IQ测试题目，但并没有度量他们在实际生活中找出问题最佳解决方案的能力。

有人会质疑传统智力构建的内容效度不够充分。基于专门研究智力的认知心理学家的研究，Robert Sternberg和他的同事发现智力应该包括从经验中学习的能力，运用元认知过程（比如计划）来提高学习的能力，以及适应环境的能力。同时，不同文化对智慧的理解不同[86]。因此智力的经典建构可能不够广泛。请参阅下面的“智力的其他方面”。

智力的其他方面

Sternberg的“智力三元理论或者成功智力理论”[85]提出了智力的三个方面：分析性，创造性和实践性（见图6-6）。该理论根据个人标准和社会文化背景描述了智力竞争力。成功是所有三个方面的综合运用，并在这三个方面中利用自己的长处，弥补自己的弱点。该理论涵括前面“智力因素”中的概念。

- 分析性智力：智力与信息处理内部世界之间的关系。这方面有三个组成部分。
  - ❑ 元成分

- 规划和监测其他成分使用的高阶过程。
  - ❑ 执行成分
- 执行元成分决定的低阶认知过程，类似GMA。
  - ❑ 知识习得成分
  - ❑ 在一开始学习如何解决问题的认知过程。
- 创造性智力：智力与经验的关系。这一方面说明以往的经验和新的经验如何与分析部分交互，包括内容如下。
  - ❑ 处理新事物的能力
- 在新的概念系统中学习思考的能力。
  - ❑ 自动信息处理
- 熟悉一项任务。
- 实践性智力：智力与外部世界的关系。这包括：
  - ❑ 适应自身环境；
  - ❑ 形成自身环境；
  - ❑ 选择一个不同的环境。

还有许多其他智力模型，见Sternbeng[86]的报告。

6

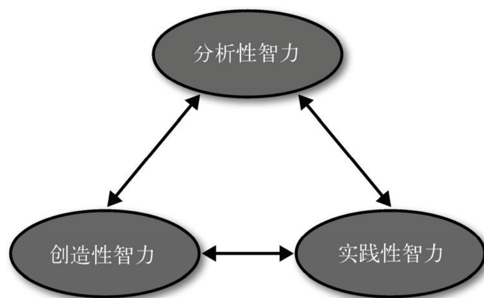


图6-6 智力的三元理论

根据Sternberg的理论，智力充分体现在生存能力和适应能力上。适应行为包括解决实际问题的能力，口头表达能力和社会竞争力。后者包括接受他人的本质，承认错误以及显示对广义世界的兴趣等。这些对协作来说都是无价的东西。

适应性行为也是Gerd Gigerenzer和“适应性行为和认知研究小组”的信条。当面对复杂的任务时，西方科学和工程学科通常教我们如何分析并获得所有相关因素，之后采取适当的行动。然而，许多任务，尤其对于没有明确定义的任务，如软件工作量估算，我们无法得到所有相关因素然后进行全面彻底的分析。凭借我们投入的努力，我们可能认为得到了很多相关因素，但我们也很可能错失了更多的因素。因此，有观点认为，这种分析方法是必然达不到目的的。人类已经在复杂环境下适应并生存了下来，靠的不是彻底分析，而是剔除无关因素，关注少数的重点[29] [30]。因此，适应

性行为就是承认缺点，并选择相应的策略。这也适用于软件开发和软件工作量估算，因为无法事先确定和分析一切。关注几个最关键因素的做法包括类比推理和自上而下的估算[61] [47] [43]。

6.3 结束语

这本书的读者应该会喜欢“反思型从业者观点”[46] [3]，从业者反思他们在日常工作中的做法，并制定更好的方法来执行工作任务。掌握从业人员通常的隐含理解，应该是像我们这样学科的主要工作重点。我必须提到，我们之前说起的人事经理也声明，她并不希望公司里面的程序员花太多时间反思他们做完的事情。相反，他们只要去执行他们被要求执行的任务就够了。我能肯定，我们中的大多数人，包括Joel Spolsky，并不喜欢这个观点，但在时间为导向的巨大压力下，很容易回复到这种策略。也就是说，因为身陷于“紧急且重要”的任务，而牺牲了“不紧急但重要”的计划和开发任务[20]。

然而，按照我们的观点，我们给受时间所困的人力资源经理设了一个逻辑陷阱：她之前的智商论点涉及GMA预测编程性能，但GMA首先预测学习能力，这就是说，她喜欢学习速度快的程序员，学习是通过反思实践完成的，但一旦她有了程序员，她不希望他们反思，因此她不希望他们学习。我把这称为不思考的、被时间所困的人事经理的悖论。

事实上，这位人力资源经理并不是完全不思考的。她知道智力起一定作用。也许她已经阅读了这章所引用的一些文献。但她或许没有读到智力有益于获取知识，而不是预测已知的技能。智力不仅仅指智商（智商只是智力的一方面）。阅读科技文献时很容易遗失细节，这不会是第一次。软件工程里有这样一个传说：Royce在他的文章[70]中反对瀑布模型，但业界未能翻到下一页，在那里他说道，前一页的模型是不被推荐。这样一知半解的例子在从业者和研究者中一定还有很多。当在基于证据做出决定时，请先更全面地了解你的证据。

让我们回到本章开始提出的三个问题。第一个问题有关是否可以定义怎样是优秀的软件开发人员。对于软件开发的某些任务（例如，编程），我们可能很快就会得出一个定义。对于其他任务（例如，软件工作量估算），则很难定义。你也许会反对说，一个优秀的工作量估算者不就是以正确的信心指数准确可靠地做出估算吗。但在这之前，请记住我们这一章所赋予第一个问题的意义远远不止于此。定义一个出色表现者意味着把任务和任务所需的专业技能定义到如下的程度：新手怎样才能成为专家，专家怎样才能成为更好的专家。

对第二个问题的回答与第一个问题类似。对于某些任务，我们能度量专业技能和任务难度，但对其他任务我们还不能这么说。请注意，我们需要有效的方法来确定一定程度的专业技能，即：无需观察很长一段时间的表现。因此，这包括预测未来的表现。

本章所涉及的两个主要任务——编程和软件工作量预估本质上很不同：一个涉及规划，另一个涉及执行。而我们对前两个问题的答案似乎扩大了这一鸿沟。但是仅仅因为我们开始对一个任务有了解，而对另一个任务没有了解，并不意味这两个任务无关。相反，人们很自然地会询问，它们之间的表现会不会有相互联系。

当前说来，编程技能、预测的准确度、完成自身任务的可靠度是正相关的，这与人们通常认

为的“即使是好的程序员也不能估算自己的工作量”的认识相反。如果正相关性被证明是真实的，你就可以让你最好的程序员估算工作量，并根据实际完成这项工作的程序员的水平来校正预估。相对当前使用的与技能无关的补偿因素来说，这是一个巨大的进步。这里使用一个概念替代另一个概念的方法，是一个有趣的前景。例如，你可以通过提高编程技能来间接提高估算的准确度吗？

第三个问题，如果不能可靠地确定专业技能，是否应侧重于工具。如果能明白任务和所需的专业技能，那么我们肯定应该更多地专注于专业技能。因为大量的研究表明，增加组织机构几个百分点的专业技能，就能提高它的商业价值。但显然，Glass的观点只有在你知道如何识别程序员的专业技能时才会起效。当你不了解专业技能，以及不知道如何发展它时，另一种方法就是依靠环境和发展辅助环境的工具和技术。然后相关技能就变成了对于辅助环境的工具和技术的掌握。可能这是把“定义不明”的任务（比如软件工作量估算）至少转换为“不稳定”任务（如果做不到“稳定”的话）的方法。

6.4 参考文献

[1] [Ackerman and Beier 2006] Ackerman, P.L., and Beier, M.E. 2006. Methods for studying the structure of expertise: Psychometric approaches. In *The Cambridge Handbook of Expertise and Expert Performance*, ed. K.A. Ericsson, N. Charness, P.J. Feltovich, and R.R. Hoffman, 147-166. New York: Cambridge University Press.

[2] [Acuña et al. 2006] Acuña, S.T., N. Juristo, and M. Moreno. 2006. Emphasizing human capabilities in software development. *IEEE Software*, 23(2): 94-101.

[3] [Argyris and Schön 1996] Argyris, C., and D.A. Schön. 1996. *Organizational Learning II. Theory, Method, and Practice*. Boston: Addison-Wesley.

[4] [Arisholm and Sjøberg 2004] Arisholm, E., and D.I.K. Sjøberg. 2004. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*. 30: 521-534.

[5] [Arisholm et al. 2007] Arisholm, E., H. Gallis, T. Dybå, and D.I.K. Sjøberg. 2007. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*. 33: 65-86.

[6] [Barrick et al. 2001] Barrick, M.B., M.K. Mount, and T.A. Judge. 2001. Personality and performance at the beginning of the new millennium: What do we know and where do we go next? *International Journal of Selection and Assessment* 9(1/2): 9-30.

[7] [Bell 2007] Bell, S.T. 2007. Deep-level composition variables as predictors of team performance: A meta-analysis. *Journal of Applied Psychology* 92(3): 595-615.

[8] [Bergersen 2010a] Bergersen, G.R. 2010. *Assessing Programming Skill*, to be submitted in 2010. PhD thesis, Simula Research Laboratory/University of Oslo.

[9] [Bergersen 2010b] Bergersen, G.R. 2010. Combining time and correctness in the scoring of performance on items. Presentation at Rasch Conference 2010, Probabilistic models for measurement in education, psychology, social science and health, June 13-16, in Copenhagen, Denmark. *https://conference.cbs.dk/index.php/rasch/Rasch2010/paper/view/596.*

[10] [Bergersen and Gustafsson 2010] Bergersen, G.R., and J.-E. Gustafsson. Programming skill, knowledge and working memory among software developers from an investment theory perspective. Forthcoming in *Journal of Individual Differences*.

[11] [Bond 1995] Bond, L., 1995. Unintended consequences of performance assessment: Issues of bias and fairness. *Educational Measurement: Issues and Practice* 14(4): 21-24.

[12] [Børte and Nerland 2010] Børte, K., and M. Nerland. Software effort estimation as collective accomplishment: An analysis of estimation practice in a multi-specialist team. To appear in *Scandinavian Journal of Information Systems*.

[13] [Campbell et al. 1993] Campbell, J.P., R.A. McCloy, S.H. Oppler, and C.E. Sager. 1993. A theory of performance. In *Personnel Selection in Organizations*, ed. N. Scmitt and W.C. Borman, 35-70. San Francisco: Jossey-Bass.

[14] [Capretz 2003] Capretz, L.F. 2003. Personality types in software engineering. *Journal of Human-Computer Studies* 58: 207-214.

[15] [Capretz and Ahmed 2010] Capretz, L.F., and F. Ahmed. 2010. Making sense of software development and personality types. *IT Professional* 12(1): 6-13.

[16] [Carroll 1993] Carroll, J.B. 1993. *Human Cognitive Abilities: A Survey of Factor-Analytic Studies*. New York: Cambridge University Press.

[17] [Cegielski and Hall 2006] Cegielski, C.G., and D.J. Hall. 2006. What makes a good programmer? *Communications of the ACM* 49(10): 73-75.

[18] [Chi 2006] Chi, M.T.H. 2006. “Two approaches to the study of experts’ characteristics.” In *The Cambridge Handbook of Expertise and Expert Performance*, ed. K.A. Ericsson, N. Charness, P.J. Feltovich, and R.R. Hoffman, 21-30. New York: Cambridge University Press.

[19] [Costa and McCrae 1985] Costa, P.T., and R.R. McCrae. 1985. *The NEO Personality Inventory Manual*. Lutz, FL: Psychological Assessment Resources.

[20] [Covey et al. 1999] Covey, S.R., A. Roger Merill, and R.R. Merill. 1999. *First Things First*. New York: Simon & Schuster.

[21] [Devito Da Cunha and Greathead 2007] Devito Da Cunha, A., and D. Greathead. 2007. Does personality matter? An analysis of code-review ability. *Communications of the ACM* 50(5): 109-112.

[22] [Dick and Zarnett 2002] Dick, A.J., and B. Zarnett. 2002. Paired programming and personality traits. *Proc. Third Int’l Conf. Extreme Programming and Agile Processes in Software Engineering (XP2002)*: 82-85.

[23] [Dickson and Kelly 1985] Dickson, D.H., and I.W. Kelly. 1985. The “Barnum Effect” in personality assessment: A review of the literature. *Psychological Reports* 57: 367-382.

[24] [Dybå et al. 2007] Dybå, T., E. Arisholm, D.I.K. Sjøberg, J.E. Hannay, and F. Shull. 2007. Are two heads better than one? On the effectiveness of pair programming. *IEEE Software* 24(6): 12-15.

[25] [Ericsson 2006] Ericsson, K.A. 2006. “An introduction to The Cambridge Handbook of Expertise and Expert Performance: Its development, organization, and content.” In *The Cambridge Handbook of Expertise and Expert Performanc*e, ed. K.A. Ericsson, N. Charness, P.J. Feltovich, and R.R. Hoffman, 3-20. New York: Cambridge University Press.

[26] [Fallaw and Solomonsen 2010] Fallaw, S.S., and A.L. Solomonsen. 2010. Global assessment trends report. Technical report, PreVisor Talent Measurement.

[27] [Forer 1949] Forer, B.R. 1949. The fallacy of personal validation: A classroom demonstration of gullibility. *Journal of Abnormal and Social Psychology* 44: 118-123.

[28] [Furnham 1996] Furnham, A. 1996. The big five versus the big four: The relationship between the Myers-Briggs Type Indicator (MBTI) and NEO-PI five factor model of personality. *Personality and Individual Differences* 21(2): 303-307.

[29] [Gigerenzer 2007] Gigerenzer, G. 2007. *Gut Feelings: The Intelligence of the Unconscious*. New York: Viking.

[30] [Gigerenzer and Todd 1999] Gigerenzer, G., and P.M. Todd, editors. 1999. *Simple Heuristics That Make Us Smart*. New York: Oxford University Press.

[31] [Glass 2002] Glass, R.L. 2002. *Facts and Fallacies of Software Engineering*. Boston: Addison-Wesley Professional.

[32] [Goldberg 1990] Goldberg, L.R. 1990. An alternative description of personality: The big-five factor structure. *Journal of Personality and Social Psychology* 59: 1216-1229.

[33] [Goldberg 1993] Goldberg, L.R. 1993. The structure of phenotypic personality traits. *American Psychologist* 48: 26-34.

[34] [Goldberg et al. 2006] Goldberg, L.R., J.A. Johnson, H.W. Eber, R. Hogan, M.C. Ashton, C.R. Cloninger, and H.C. Gough. 2006. The international personality item pool and the future of public-domain personality measures. *Journal of Research in Personality* 40: 84-96.

[35] [Grimstad and Jørgensen 2007] Grimstad, S., and M. Jørgensen. 2007. Inconsistency in expert judgment-based estimates of software development effort. *Journal of Systems and Software* 80(11): 1770-1777.

[36] [Gruschke and Jørgensen 2005] Gruschke, T.M., and M. Jørgensen. 2005. Assessing uncertainty of software development effort estimates: Learning from outcome feedback. *Proceedings of the 11th IEEE International Software Metrics Symposium*: 4.

[37] [Hærem 2002] Hærem, T. 2002. *Task Complexity and Expertise as Determinants of Task Perceptions and Performance: Why Technology-Structure Research has been Unreliable and Inconclusive*. PhD thesis, Norwegian School of Management BI.

[38] [Halkjelsvik et al. 2010] Halkjelsvik, T., M. Jørgensen, and K.H. Teigen. 2010. To read two pages, I need 5 minutes, but give me 5 minutes and I will read four: How to change productivity estimates by inverting the question. Forthcoming in *Applied Cognitive Psychology*.

[39] [Hannay et al. 2007] Hannay, J.E., D.I.K. Sjøberg, and T. Dybå. 2007. A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering.* 33: 87-107.

[40] [Hannay et al. 2009] Hannay, J.E., T. Dybå., E. Arisholm, and D.I.K. Sjøberg. 2009. The effectiveness of pair programming: A meta-analysis. *Information & Software Technology* 55(7): 1110-1122.

[41] [Hannay et al. 2010] Hannay, J.E., E. Arisholm, H. Engvik, and D.I.K. Sjøberg. 2010. Personality and pair programming. *IEEE Transactions on Software Engineering* 36(1): 61-80.

[42] [Hanson and Claiborn 2006] Hanson, W.E., and C.D. Claiborn. 2006. Effects of test interpretation style and favorability in the counseling process. *Journal of Counseling and Development* 84(3): 349-358.

[43] [Hertwig et al. 1999] Hertwig, R., U. Hoffrage, and L. Martignon. 1999. Quick estimation: Letting the environment do the work. In *Simple Heuristics That Make Us Smart*, ed. G. Gigerenzer and P.M. Todd, 75-95. New York: Oxford University Press.

[44] [Hogan et al. 2000] Hogan, K., B.K. Nastasi, and M. Pressley. 2000. Discourse patterns and collaborative scientific reasoning in peer and teacher-guided discussions. *Cognition and Instruction,* 17(4): 379-432.

[45] [Horn and Masunaga 2006] Horn, J., and H. Masunaga. 2006. A merging theory of expertise and intelligence. In *The Cambridge Handbook of Expertise and Expert Performance,* ed. K.A. Ericsson, N. Charness, P.J. Feltovich, and R.R. Hoffman, 587-612. New York: Cambridge University Press.

[46] [Jarvis 1999] Jarvis, P. 1999. *The Practitioner-Researcher*. San Francisco: Jossey-Bass.

[47] [Jørgensen 2004] Jørgensen, M. 2004. Top-down and bottom-up expert estimation of software development effort. *Information and Software Technology* 46(1): 3-16.

[48] [Jørgensen 2005] Jørgensen, M. 2005. Practical guidelines for expert-judgment-based software effort estimation. *IEEE Software* 22(3): 57-63.

[49] [Jørgensen 2010] Jørgensen, M. 2010. Selection of strategies in judgment-based effort estimation. *Journal of Systems and Software* 83(6): 1039-1050.

[50] [Jørgensen and Carelius 2004] Jørgensen, M., and G.J. Carelius. 2004. An empirical study of software project bidding. *IEEE Transactions on Software Engineering* 30(12): 953-969.

[51] [Jørgensen and Grimstad 2008] Jørgensen, M., and S. Grimstad. 2008. Avoiding irrelevant and misleading information when estimating development effort. *IEEE Software* 25(3): 78-83.

[52] [Jørgensen and Grimstad 2010] Jørgensen, M., and S. Grimstad. 2010. The impact of irrelevant and misleading information on software development effort estimates: A randomized controlled field experiment. *IEEE Transactions on Software Engineering* Preprint(99).

[53] [Jørgensen and Halkjelsvik 2010] Jørgensen, M. and T. Halkjelsvik. 2010. The effects of request formats on judgment-based effort estimation. *Journal of Systems and Software* 83(1): 29-36.

[54] [Jørgensen and Sjøberg 2001] Jørgensen, M., and D.I.K. Sjøberg. 2001. Impact of effort estimates on software project work. *Information and Software Technology* 43(15): 939-948.

[55] [Jørgensen and Sjøberg 2004] Jørgensen, M., and D.I.K. Sjøberg. 2004. The impact of custom mer expectation on software development effort estimates. *Journal of Project Management* 22(4): 317-325.

[56] [Jørgensen et al. 2004] Jørgensen, M., K.H. Teigen, and K.J. Moløkken-Østvold. 2004. Bettersure than safe? Over-confidence in judgment based software development effort prediction intervals. *Journal of Systems and Software* 70(1-2): 79-93.

[57] [Kahneman and Frederick 2004] Kahneman, D., and S. Frederick. 2004. “A Model of Heuristic Judgment” in *The Cambridge Handbook of Thinking and Reasoning*, eds. K.J. Holyoak and R.G. Morrison. Cambridge: Cambridge University Press, 267-294.

[58] [Karn and Cowling 2005] Karn, J.S., and A.J. Cowling. 2005. A study of the effect of personality on the performance of software engineering teams. *Proc. Fourth International Symposium on Empirical Software Engineering* (ISESE’05): 417-427.

[59] [Karn and Cowling 2006] Karn, J.S., and A.J. Cowling. 2006. A follow up study of the effectof personality on the performance of software engineering teams. *Proc. Fifth International Symposium on Empirical Software Engineering* (ISESE’06): 232-241.

[60] [LeBoeuf and Shafir 2004] LeBoeuf, R.A., and E.B. Shafir. 2004. “Decision Making” in *The Cambridge Handbook of Thinking and Reasoning*, eds. K.J. Holyoak and R.G. Morrison. Cambridge: Cambridge University Press, 243-266.

[61] [Li et al. 2007] Li, J., G. Ruhe, A. Al-Emran, and M.M. Richter. 2007. A flexible method for software effort estimation by analogy. *Empirical Software Engineering* 12(1): 1382-3256.

[62] [Meehl 1956] Meehl, P.E. 1956. Wanted—A good cookbook. *American Psychologist* 11(3): 263-272.

[63] [Moløkken-Østvold and Jørgensen 2003] Moløkken-Østvold, K.J., and M. Jørgensen. 2003. A review of surveys on software effort estimation. *Proc. Int'l Symp. Empirical Software Engineering* (ISESE 2003): 223-230.

[64] [Moløken-Østvold and Jørgensen 2005] Moløkken-Østvold, K.J., and M. Jørgensen. 2005. A comparison of software project overruns—flexible vs. sequential development models. *IEEE Transactions on Software Engineering* 31(9): 754-766.

[65] [Moore 1991] Moore, J.E. 1991. Personality characteristics of information systems professionals. *Proc. 1991 Special Interest Group on Computer Personnel Research* (*SIGCPR*) *Annual Conference*: 140-155.

[66] [Paul 2005] Paul, A.M. 2005. *The Cult of Personality Testing: How Personality Tests Are Leading Us to Miseducate Our Children, Mismanage Our Companies, and Misunderstand Ourselves*. New York: Free Press.

[67] [Peeters et al. 2006] Peeters, M.A.G., H.F.J.M. van Tuijl, C.G. Rutte, and I.M.M.J. Reymen. Personality and team performance: A meta-analysis. *European Journal of Personality* 20: 377-396.

[68] [Pervin and John 1997] Pervin, L.A., and O.P. John. 1997. *Personality: Theory and Research*, Seventh Edition. Hoboken, NJ: John Wiley & Sons.

[69] [Read et al. 2010] Read, J.S., B.M. Monroe, A.L. Brownstein, Y. Yang, G. Chopra, and L.C. Miller. 2010. A Neural Network Model of the Structure and Dynamics of Human Personality. *Psychological Review* 117(1): 61-92.

[70] [Royce 1970] Royce, W.W. 1970. Managing the development of large software systems. *Proc. IEEE WESCON*: 1-9.

[71] [Saggio and Kline 1996] Saggio, A., and P. Kline. 1996. The location of the Myers-Briggs Type Indicator in personality factor space. *Personality and Individual Differences* 21(4): 591-597.

[72] [Saggio et al. 2001] Saggio, A., C. Cooper, and P. Kline. 2001. A confirmatory factor analysis of the Myers-Briggs Type Indicator. *Personality and Individual Differences* 30: 3-9.

[73] [Saleh et al. 2010] Saleh, N., E. Mendes, J. Grundy, and G. St. J. Burch. 2010. An empirical study of the effects of conscientiousness in pair programming using the five-factor personality model. *Proc. of the 32nd ACM/IEEE Int'l Conf. Software Engineering* 1: 577-586.

[74] [Schmidt and Hunter 1998] Schmidt, F.L., and J.E. Hunter. 1998. The validity and utility ofselection methods in personnel psychology: Practical and theoretical implications of 85 yearsof research findings. *Psychological Bulletin* 124(2): 262-274.

[75] [Schmidt et al. 1986] Schmidt, F.L., J.E. Hunter, and A.N. Outerbridge. 1986. Impact of job experience and ability on job knowledge, work sample performance, and supervisory ratings of job performance. *Journal of Applied Psychology* 71(3): 432-439.

[76] [Schmidt et al. 1988] Schmidt, F.L., J.E. Hunter, A.N. Outerbridge, and S. Goff. 1988. Joint relation of experience and ability with job performance: Test of three hypotheses. *J. Applied Psychology* 73(1): 46-57.

[77] [Schweizer 2005] Schweizer, K. 2005. An overview of research into the cognitive basis of intelligence. *Journal of Individual Differences* 26(1): 43-51.

[78] [Shaft and Vessey 1998] Shaft, T.M., and I. Vessey. 1998. The relevance of application domain knowledge. *Journal of Management Information Systems* 15(1): 51-78.

[79] [Shneiderman 1980] Shneiderman, B. 1980. *Software Psychology: Human Factors in Computer and Information Systems.* Cambridge, MA: Winthrop Publishers.

[80] [Smith 1989] Smith, D.C. 1989. The personality of the systems analyst: An investigation. *SIGCPR Computer Personnel* 12(2): 12-14.

[81] [Sonnentag 1998] Sonnentag, S. 1998. Expertise in professional software design. *Journal of Applied Psychology* 83(5): 703-715.

[82] [Spangler 1992] Spangler, W.D. 1992. Validity of questionnaire and TAT measures of need for achievement: Two meta-analyses. *Psychological Bulletin* 112(1): 140-154.

[83] [Spearman 1923] Spearman, C. *The Nature of "Intelligence" and the Principles of Cognition,* Second Edition. New York: Macmillan.

[84] [Spolsky 2007] Spolsky, J. 2007. *Smart and Gets Things Done*. New York: Apress.

[85] [Sternberg 2003] Sternberg, R.J. 2003. A broad view of intelligence: The theory of successful intelligence. *Consulting Psychology Journal: Practice and Research* 55(3): 139-154.

[86] [Sternberg 2005] Sternberg, R.J. 2005. Intelligence. In *The Cambridge Handbook of Thinking and Reasoning,* ed. K.J. Holyoak and R.G. Morrison, pp. 751-774. New York: Cambridge University Press.

[87] [Turley and Bieman 1995] Turley, R.T., and J.M. Bieman. 1995. Competencies of exceptional and non-exceptional software engineers. *Journal of Systems and Software* 28(1): 19-38.

[88] [Valentin Kvist and Gustafsson 2008] Valentin Kvist, A., and J.-E. Gustafsson. 2008. The relation between fluid intelligence and the general factor as a function of cultural background: A test of Cattell's investment theory. *Intelligence* 36: 422-436.

[89] [Walle and Hannay 2009] Walle, T., and J.E. Hannay. Personality and the nature of collaboration in pair programming. *Proc. 3rd Int'l Symp. Empirical Software Engineering and Measurement* (ESEM): 203-213.

[90] [Weinberg 1971] Weinberg, G.M. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

[91] [Weinberg 1998] Weinberg, G.M. 1998. *The Psychology of Computer Programming*. New York: Dorset House Publishing.

[92] [Williams et al. 2006] Williams, L., L. Layman, J. Osborne, and N. Katira. 2006. Examining the compatibility of student pair programmers. *Proc. AGILE 2006*: 411-420.

[93] [Woodruff 1979] Woodruff, C.K. 1979. Personality profiles of male and female data processing personnel. *Proc. 17th Annual Southeast Regional Conference*: 124-128.

第7章

为什么学编程这么难

Mark Guzdial

这本书的大部分内容都以某种方式围绕着程序员展开，如开发软件的最佳方式、开发软件的成本、什么样的交流能促进编程。但首要问题是，要成为程序员就很困难。试图进入该领域的人少，真正进来的更少。在这一章中，我们要问：为什么学编程这么难？

我们是否需要在当前发展更多的程序员？这是一个争议点。美国劳工统计局刚刚预言了对计算机专业人才的巨大需求。根据2008年11月的报告[4]，从2006年到2016年，对“IT专业人士”的需求增长率将是其余劳动力的两倍。2009年11月的估计更新道：“‘计算机与数学’是最快速增长的主要职业组中增长最快的职业集群。”[3]但什么是“IT专业人士”？是 “计算机与数学”的职业吗？基于许多新近失业的IT工作者的经验，美国的程序员也许已经太多了[23]，特别是在当前的经济衰退期。

虽然还不能确定我们是否需要更多的程序员，但很明确的是，许多开始走上编程之路的人很早就失败了。关于入门级计算机课程（通常用CS1指代，根据一份早期课程标准报告）高挂科率的传言在文献和会议的走廊交谈中非常普遍，这样的会议包括美国计算机协会（ACM）计算机科学教育特别兴趣小组（SIGCSE）的研讨会。Jens Bennedsen和Michael Caspersen是第一个试图合理地找出失败率的真实数据的人[2]。他们通过几个计算机科学（CS）教师的邮件列表向全世界的教员寻求数据。有63所学院提供了他们入门课程的通过率，这意味着这些数据是自我选择和自我汇报的（比如，结果实在太尴尬的学校可能选择不参与或者说谎，所以结果可能是歪曲的，因为样本并不随机）。大体上说，30%的学生在第一门课就会挂科或者退课，专科学院比综合大学的比率更高些（40%对30%）。因此，我们可以大概知道，在全世界所有机构中，每三个开始CS1课程的学生就有一个失败或者放弃。这究竟是为什么？

Bennedsen和Caspersen的结果公布的只是课程学习的成功或失败。然而，CS1老师的标准并不是成功的唯一定义。许多程序员从没有上过任何课程，但仍然很成功。所以，除了成绩的证明外，我们必须首先确定学生学习编程确实很困难。如果我们可以确定这个，下一个问题会是“为什么”。编程是一个非自然的行为吗？如果换一种方式，编程是不是会变得更简单？编程如果能以另一种方式教授，会不会学起来更简单？或者，也许我们压根就不知道如何衡量学生对编程的了解情况。

7.1　学生学习编程有困难吗

在20世纪80年代的耶鲁大学，Elliot Soloway在他的Pascal编程课程中定期布置相同的作业[28]：

　　写一段代码，重复读取正整数，直到读入99999。在看到99999之后，代码应该输出平均值。

这在计算机教育研究的初期成为了被研究最多的问题之一，被称为“降雨量问题”。在1983年用这个公式表达这个问题的论文中（其他的论文可能探究其他的公式），耶鲁团队探索了是否加一个“leave”语句（C或Python中的break语句）会提高学生在这个问题上的得分。他们把这个问题交给3个小组。

- CS1第一学期学习了3/4的学生，已经学习和使用了WHILE、REPEAT和FOR语句。
- CS2（第二学期的计算课程，常见的是数据结构课程）已经学习了3/4的学生。
- 系统编程课程中大三大四的学生。

在每个班级的学生中，一半使用传统的Pascal语言，另一半有机会使用添加了leave语句的Pascal。在表7-1中总结的数据也许会使那些已经成功进入编程行业的人感到震惊。只有14%的入门级学生可以用原始的Pascal解决这个问题？并且30%最高级的学生也不能解决？这个研究多次重复出现在文献中（比如，Jim Spohrer[29]和Lewis Johnson[15]的论文都研究了学生在降雨量问题上的工作结果），之后也非正式地重复了多次，令人吃惊的是，每次都有相似的结果。

表7-1　耶鲁学生在降雨量问题上的表现

班　　组	使用原始Pascal的准确率	使用带leave的Pascal的准确率
CS1	14%	24%
CS2	36%	61%
系统编程	69%	96%

这个问题需要一个相对复杂的条件控制循环。如果输入是负数，忽略它并继续接受输入。如果输入是正数而不是99999，把它加入总数并增加一次计数。如果输入是99999，忽略输入并退出循环。很容易把负数或者99999计入总数而出错。

这些结果来自耶鲁大学。可不可能是耶鲁大学的编程教学水平很差？那些学生中只有少数在进入大学前学习了编程，所以无论他们得到了什么指导，都是从CS1课程中来的。许多年来，研究者一直想知道如何开展一项编程的研究，能避免某一所学校可能产生的误导所引起的复杂因素。

7.1.1　2001年McCracken工作小组

在2001年，Mike McCracken[18]组织了一个研究小组，在坎特伯雷肯特大学举办的计算机科学教育创新和技术（ITICSE）大会上相聚。ITICSE是一个在欧洲举行的大会，吸引着全世界各地的人前来参与。McCracken小组的老师们想要进行与他们CS1或者CS2课程同样的研究：布置同样的问题，给学生90分钟在纸上完成任务。所有学生的数据都会提交给大会，并由大会参与者分析。3个国家的4个机构参加了这第一次“多机构，多国家”（MIMN）的计算机系学生成绩研究。通

过比较跨机构跨国家的学生，研究者希望对学生在第一年能够做什么有更清楚的概念。

学生要解决的问题是：评估只含有数字、二元运算（+，−，/，*）符或一元负运算符（~，为了避免减号的重载）的算数表达式（只包括数字和符号）。一共有216个学生提交了答案。评分与编程语言无关，总分110分，而平均分只有22.89分（21%）。学生们在这个问题上表现太差了！一位老师甚至“作弊”，他在布置问题之前先对学生直接讲解了表达式如何评估。那个班级的表现也没有好到哪里去。

McCracken工作小组对他们的数据做了几次评估。他们发现班级与班级之间表现相差悬殊。他们也看到了许多计算机老师已经注意到的、一些文章也试图解释的“双峰效应”的证据。一些学生就是能“明白”并且表现得很好。更多学生所在的那个“驼峰”却表现得差很多。为什么一些学生就是能“明白”编程而另一些不能呢？有人探究了不同的变量，从过往计算机经验到数学背景[2]，仍然没有找到对这个结果的有力解释。

7.1.2 Lister工作小组

一些学生可能对一位老师或一种教授方式没有共鸣，但是为什么不同机构的这么多学生都如此差劲？所有地方的教学都很差吗？我们高估了学生的能力吗？或者是我们没有度量正确的东西吗？Raymond Lister在2004年组织了第二次ITICSE工作小组，来探究这些问题[17]。

Lister小组的想法是McCracken小组对学生的要求太高了。给他们的问题需要对设计和实现方案的高层次思考。而Lister小组决定注重阅读和追踪代码的低层次能力。他们制作了一个选择题问卷（MCQ）让学生执行任务，如阅读代码并识别输出，或者识别代码段空白处的正确代码。问题针对数组操作。他们邀请世界各地的参与者给他们的学生同样的问卷，并把结果交给ITICSE。

Lister小组的结果稍好，但仍然令人失望。556个学生的平均分是60%。虽然这些结果确实使人想到McCracken小组高估了学生的能力，但是Lister和他的小组对他们的问卷有着更高的期望。

McCracken和Lister的工作告诉了研究者，要低估学生在第一个课程中对编程的理解很困难。很明显，他们学的比我们知道的少多了。现在问题来了，一些学生在学习，而大多数人不在。编程中什么东西这么困难，以至于大部分学生不能简单上手？

7.2 人们对编程的本能理解是什么

语言学家普遍认同，人类对语言的兴奋度很高。我们的大脑进化至今，能快速并高效地开始学习新语言，特别是对自然语言。然而，编程是一种对人工语言的操纵，是为了一个特定的、相对不自然的目的而发明的——“确切地”告诉一个非人类的代理（电脑）应该做什么。也许编程对我们来说不是一个自然的行为，只有少数人能成功地在这非自然行为上做复杂的头脑体操。

我们应该如何回答这个问题呢？可以尝试一种类似于Lister对McCracken方法的改进：选择任务的一小部分并专注于此。编程需要用非自然语言告诉机器做什么。如果我们让参与者用自然语言告诉另一个人去完成某些任务，那会如何？参与者会如何定义他们的“程序”？如果我们去除人工语言，编程会是“自然的”或者“常识性的”的吗？

L.A. Miller要求他的研究参与者创造指令让别人执行[19]。参与者得到了各种文件（如员工、工作和薪资信息）和任务的说明，如下所示。

> 制作一份符合下列标准之一的员工列表：
> (1)有技师头衔并且每小时工资在6美元以上；
> (2)未婚并且每小时工资在6美元以下。
> 列表应该根据员工姓名组织起来。

Miller弄清楚了对于参与者来说什么是复杂的和什么是容易的。首先，他的研究对象都完成了任务。并没有像编程课中发生的那样，三分之一的人放弃或失败。为别人指定程序的这项基本挑战看来并不成问题。

Millier问题的自然语言解决方案和之前研究者研究的编程问题有一个关键的不同点：解决方案的结构。Miller研究对象没有定义迭代，而是设定了操作。举例说来，他们没有说“找到每个文件夹查看姓氏。如果以G开头……”而是说关于如何做事“对所有姓氏以G开头的人……”令Miller感到惊讶的是：没有一个人对他们的循环设定了结束条件。一些人说了类似于IF的可测试条件，但没人用ELSE。单从这个结果就可以看出，定义一个对初学者更自然的编程语言是可能的。

Miller另外做了一个实验，他把第一个实验中有含糊循环的指令给其他参与者。没有人对此指令有任何的问题。很明显，当你处理完数据后就会停下来。研究对象处理完了数据组，却没有增加任何索引。

John Pane在20世纪90年代末和21世纪初继续了这项探索[22]。Pane对创造一种编程语言非常感兴趣，能更贴近于人们对他人描述流程的“自然”方式。他复制了Miller的实验，但是任务和输入不同。他担心Miller提供“文件”并要求“列表”的方式可能诱导了参与者。取而代之，Pane向研究对象展示了图片和游戏片段，然后询问他们会如何告诉电脑从而使那些事情发生。比如，“写一段话，总结我（作为电脑）应该如何针对其他事物的存在或空缺而移动‘吃豆人’。”

和Miller一样，Pane发现人们不会用循环。他从编程范式的角度进一步辨别了指令种类。他发现有很多人使用了约束（“这个东西总是做那个事情”）、事件驱动编程（“当‘吃豆人’吃到所有豆时，他会升级到下一个级别”）和命令编程。没有人哪怕一次提到“对象”。他们说到游戏中实体的特征和行为，但不对那些实体分组（如，归入“类”）。没有人从实体本身的视角谈论行为，所有事情都是从玩家或者程序员的角度出发的。

基于Miller和Pane的实验，我们也许能声称：人有能力对另一个代理指定任务，但是我们当前的编程语言不允许人们以他们思考任务的方式来编程。如果编程语言做得更自然，大部分学生是不是就能够编程了？人们能用更自然的语言解决牵涉重要算法的复杂问题吗？更自然的语言是不是对新手的任务和专业人员的任务均有益处？如果不是，CS的学生们是不是仍然需要在某个时间点开始学习专业语言？

一批自称为“常识计算小组”的研究者已经问过了这些问题中的一部分。他们要求CS1的预备学生在学习编程语言之前用自然语言解决重要的算法任务，比如排序或并行进程。他们发现研

究对象在这些任务上的成功率惊人。

在一项研究中[16]，他们要求学生为影院创造一套流程，该影院决定使用两个票务销售商。

> 假设我们通过电话用以下方式销售音乐会门票：当一位顾客打入电话要求n个座位时，销售者第1步找到n个最佳空余座位，第2步把这些座位标记为有人，第3步处理顾客的付费选项(如,得到一个信用卡或者借记卡的卡号,或者把票寄给预订售货窗口自取)。
>
> 假设有不止一个销售电话同时工作。我们可以预见什么问题？应该如何避免这些问题？

来自5个机构的约66个参与者试图解决这个问题，成功率惊人！如表7-2所示，几乎所有的学生都找出了问题真正的难点，其中71%提出了可工作的解决方案。大部分方案效率不高，因为涉及一个集中仲裁器，所以这些学生仍然需要学习很多东西。然而，他们能解决并行处理问题的这一事实，表明了阻碍学生学会编程的问题可能更多地出在工具上。学生的计算思维可能比我们想象的更强。

表7-2 学生识别出的问题数和解决数（n=66），摘自参考文献[16]

成　就	学生完成率
问题识别	
• 一张票不止卖了一次	97%
• 其他	41%
为并发问题提供“合理的”解决方案	71%

7.3 通过可视化编程来优化工具

如何优化工具？一个明显的答案是转移至更可视的图标系统。自从David Smith基于图标的编程语言Pygmalion的出现[27]，理论上就一直认为可视化思维对学生更容易。当然也有很多研究表明可视化大体上能帮助学生学习计算机[21]，但谨慎的研究相对较少。

之后，Thomas Green和Marian Petre在类数据流编程语言和文本编程语言之间，做了一个针尖对麦芒的比较[17]。他们用两种可视化语言创造了两个程序，这两种语言在之前的研究中被证实工作良好。他们也创造了一个文本语言程序，该语言同样测试良好。研究者在短时间内向研究对象展示一段可视化程序或者文本程序，然后问一些关于程序的问题（如，被展示的输入数据和输出结果）。*理解图形化语言总是花费更多的时间*。研究对象有多少可视化或文本语言的经验，有哪种可视化语言的经验，这些都没有关系。研究对象理解可视化语言总比理解文本语言更慢。

Green和Petre基于这项研究的变种发布了若干篇文章[8][9]。但是真正的试验来自于Tom Moher和他的同事们[20]，他们为了支持可视化语言而暗中作弊。Tom和他手下的研究生们使用一种可视化图标系统来教导高中学生编程，叫作Petri Nets。他得到了一份Green和Petre的材料副本，并创造了一个只用Petri Nets作为可视化语言的版本。然后Tom把*他自己和他的学生们*作为研究对象重新运行了这项研究。使人吃惊的是，文本语言再一次被证实更容易理解，在每个条件下都如此。

我们对可视化语言的直觉是错误的吗？可视化是否实际上降低了人们对软件的理解力？那Naps他们谈到的那些研究又算什么[21]？他们错了吗？

有一个对比多项研究的标准方法，叫做*元研究*。Barbara Kitchenham在本书的第3章描述了这一程序。Chris Hundhausen, Sarah Douglas和John Stasko对算法可视化的研究做了同样的分析[13]。他们发现确实有许多研究表明算法可视化对学生有极大的好处。但是许多研究并没有重大结果。即使有重大结果，也没有明显地展示算法可视化是*如何*起作用的（图7-1）。Hundhausen和他的同事们发现*如何*使用可视化非常重要。例如，在讲课演示中使用可视化对学生学习的影响甚微。但如果让学生构造自己的可视化工具，就会对学生的学习产生巨大的影响。

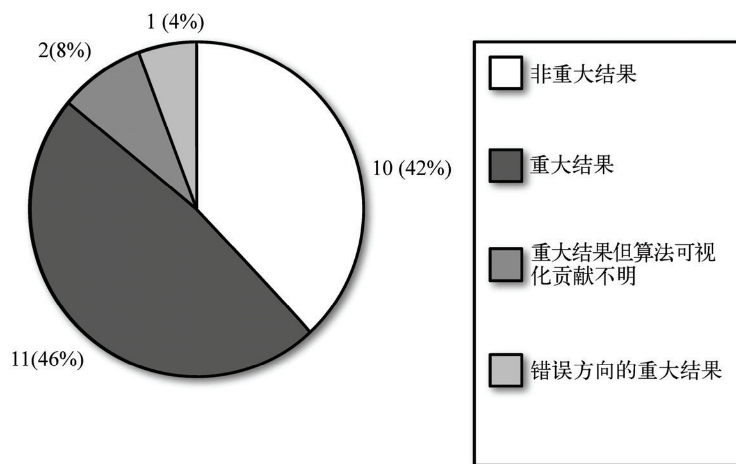


图7-1　Hundhausen, Douglas和 Stasko 的论文对的24个研究报告的总结[13]

一些研究只改变了可视化的使用方法，同时保持了其他各个变量不变（例如班级的类型、学生的类型、教师、主题等）。虽然Hundhausen和他的同事在分析了24个研究后，对可视化使用方法的重要性产生了怀疑，但是*验证猜疑*与之并不相同。我们从教学研究中学到的一件事是，预测输出其实非常困难。人并不像导弹发射或鸡尾酒调配那样可预测。事实上，我们需要测试怀疑和假设，有时需要在不同情况下反复测试，直到我们对测试结果信服。

7.4　融入语境后的改变

目前我们知道的是，开始学习计算机的学生所学到的设计和编码比我们所预期的少得多。并且，第一门课的挂科率非常高。我们发现，从文本转至可视化不一定能得到更好的结果。这表明，有其他变量发挥了作用。之前的章节探索了可视化使用方法的不同可能是一个变量。我们应该掌控哪些其他变量来提高学生的学习成功率呢？

1999年，乔治亚理工学院决定要求所有的本科生参加计算机入门课程。在开始的前几年，只有一门课满足了这一需求。这门课的总体通过率为78%，参照Bennedsen和Caspersen的分析[2]，这

已经很好了。然而，这个数字分解来看并不如意。文科、建筑、管理学院学生的通过率不到50%[31]。女生的挂科率几乎是男生的两倍。这是一门面对所有学生的课程，但是大多数工科专业的男生都通过了，这突出了计算机教学的普遍问题。

2003年，我们开始了一项试验，对那些班级的学生教授一种不同的入门课程，将媒体操作作为课程背景[6]。本质上来说，学生与其他计算机科学入门课程一样学习编程。事实上，我们努力涵盖当前ACM和IEEE标准[14]所推荐的所有主题[10]。然而，所有课本上的例子，授课中的代码实例，以及课后作业，都要求学生把数字媒体作为这些程序中的操作数据。比如，学生通过把一张图片中所有的像素都转灰，学习如何对数组的元素进行迭代；通过连接声音缓存做数字接合，来代替连接字符串；通过去除红眼而不影响图片中别的红色区域的操作，来学习对数组子区域的迭代。

这次试验得到了引人瞩目的积极反响。学生认为新的课程与生活更相关且更具信服力。特别是女生的合格率赶超了男生（但是没有任何统计意义）[24]。接下来三年的平均通过率上升至85%。而且对那些之前通过率小于50%的专业也同样有效[30]。

听起来像是一个巨大的成功，但是这些论文究竟说了什么？新方法是所知的唯一改变吗？也许那些学校突然开始接受更聪明的学生。也许乔治亚理工学院雇佣了一个有魅力的新教师，学生因为为他倾倒而开始关心课程内容。社会学研究者把这些阻止我们从研究中断言我们所希望的结论的因素称为*有效性的威胁*①。为了捍卫研究的价值，我们可以声明，文章[30]包括了不同讲师的多个学期，涵盖了3年的有价值结果，学生不太可能在突然之间变得非常不同。

即使我们有信心做出结论说，乔治亚理工学院的成功是引入媒体计算的结果，与其他关于学生和教学的因素物无关，我们也应该思考我们可以断言什么。乔治亚理工学院是一所很好的学校。他们能吸引聪明的学生，也能雇佣并留住好老师。如果你的学校引入媒体计算作为计算机入门课程，能成功吗？

第一次尝试在另一类学校做媒体计算的是乔治亚州盖斯威尔州立大学的Charles Fowler。盖斯威尔州立大学是一所两年制（没有本科生和研究生）公立学校。研究结果在同一篇论文[30]中做了汇报。Fowler也在他的学生中发现了成功率的急剧提升。Fowler的学生的跨度从计算机科学到护理学院。然而，乔治亚理工学院和盖斯威尔大学都是白人学生居多。这种方法对其他种族的学生也适用吗？

在伊利诺伊大学芝加哥分校（UIC），Pat Troy和Bob Sloan在他们的“CS 0.5”课程中引入了媒体计算[26]。他们的课程是为想学习计算机专业，但没有编程基础的学生准备的。CS 0.5是为了他们的第一门课程（CS1）做准备。在多个学期中，这些学生的合格率也得到了提升。UIC的学生包含了更多种族，大部分学生都属于少数族裔。

现在，你确信应该对学生使用媒体计算了吗？你也许会争辩说，这些仍然是不寻常案例。乔治亚理工和盖斯威尔研究的是非主修学生（盖斯威尔也有主修学生）。虽然Troy和Sloan处理了想要主修计算机科学的学生，但他们的课程并不是针对主修计算机科学本科生的普通入门课程。

① http://www.creative-wisdom.com/teaching/WBI/threat.shtml。

Beth Simon和她在加州大学圣地亚哥分校（UCSD）的同事们在两年前开始使用媒体计算作为CS主修学生的主要入门课程（“CS1”）[25]。更多的学生通过了新课程。更重要的是，上过媒体计算的学生在媒体计算后的第二门课程中，也比传统CS1课程的学生表现得更好。

确信了吗？媒体计算是一次灌篮扣杀吗？所有人都应该使用媒体计算嘛？虽然我的出版商希望你们相信这个[11][12]，但是这些研究并没有清楚地证实它。

首先，我没有声称学生在媒体计算中学到了等量的东西。如果有任何人告诉你，学生用他们的方法学到了与其他方法等量的东西，请保持强烈怀疑，因为我们目前对计算机入门课程的学习并没有可靠并有效的度量。Allison Tew在2005年第一个试图回答学生是否在不同的CS1课程中学习了同样的东西[31][30]。她开发了两套选择题测试作比较，每套针对一种CS1语言。她想让它们成为“同构的”：一个评估特定概念的问题（如，对数组的迭代）本质上对所有的测试和语言是相同的。她把这套测试用在第二门CS课程（CS2）之前和之后，从而度量学生学习CS1课程的差异。Tew发现，学生确实在他们的CS1课程中学到了不同的东西（由CS2之前的测试度量来看），但是这些不同在CS2课程结束时消失了。这是一个重大的发现，表明CS1的区别对未来的成功并不那么重要。但在以后的尝试中，她再也没有找到相同的结果。

怎么会这样？一个可能性是她的测试不完全相同。学生可能对之有不同的理解。这些测试所度量的可能不完全是她想测试的那种学习。比如，一些选择题的答案可以被猜测，因为错误的选择完全不实际以至于学生可以在不知道正确答案的情况下排除它们。一个好的测量学习的测试应该是可靠并有效的。换句话说，它度量正确的事情，而且所有学生在所有时间都对它有相同的理解。

在写这篇文章时，还没有与语言无关地、可靠并有效地对CS1学习的度量。Tew正在测试一个度量。但是直到它存在之前，还不可能确定学生是否在不同的方法下学到了同样的东西。学生在媒体计算中的成功率更高，这很好。UCSD的学生在第二门课中也学得更好，这也很好。但是我们真的不能断定学生学到了同样的东西。

其次，即使乔治亚理工、盖斯威尔、UIC和UCSD都能证明学生在入门课程中学习了等量的知识，那又能证明什么？课程对所有人都适用？不论“传统”课程如何非凡或成功，它都比“传统”课程要好？无论学生多么准备不足或者不感兴趣，它对所有学生都适用？也无论老师教得多差？那当然很荒唐。我们总能想象出事情出错的情况。

总的说来，对课程的改进方法为我们提供了处方性模式，但不提供预言性的理论。媒体计算研究向我们展示了证据，说明对于不同种类的的学生和老师，入门课程的通过率都可以被提高，但不意味着一定会提高。能承诺改进的就是处方。它不是预言性的理论——可以预计改善，因为如果不知道是否有更多还未在学生身上验证过的变量，就不能预计改善。它也不可能成为预言性的，因为我们不能说不使用媒体计算就保证会失败。

7.5　总结：一个新兴的领域

计算教学研究，作为一个研究的领域，仍然是一个新兴的学科[5]。我们最近才意识到计算教

学的重要性以及支持计算教育的需要。支持计算教师的ACM组织——计算机科学教师联盟（CSTA）——在2005年才成立。作为对比，全国数学教师委员会在1920年就成立了。

我们大多数的研究更多地指向人们学习编程有多复杂。我们持续对人类的智慧和创造力感到惊奇，即使没有受过计算机培训的学生也已经能用算法思考。然而，开发这种技能到一种程度，从而可以用机器的语言给机器下指令，比我们期待的要慢很多。我们可以帮助学生走过这一过程，但我们仍然没有有效的方法度量他们所学的多少。

我们真正在这一领域需要的是预言性理论，基于人们如何真正建立对计算的理解的模型。在这些理论之上，我们可以构造我们有信心的课程。ACM的国际计算教育研究研讨会至今只有5岁，参会人数从未超过100人。我们只有很少的工作者，他们才刚刚开始一项艰巨的任务。我们对理解为什么学生学习编程这么困难只迈出了第一步。

7.6　参考文献

[1] [Bennedsen and Caspersen 2005] Bennedsen, J., and M.E. Caspersen. 2005. An investigation of potential success factors for an introductory model-driven programming course. *Proceedings of the first international workshop on computing education research:* 155-163.

[2] [Bennedsen and Caspersen 2007] Bennedsen, J., and M.E. Caspersen. 2007. Failure rates in introductory programming. *SIGCSE Bull.* 39(2): 32-36.

[3] [Computing Community 2010] Computing Community Consortium. 2010. Where the jobs are…. *http://www.cccblog.org/2010/01/04/where-the-jobs-are/.*

[4] [Computing Research 2008] Computing Research Association. 2008. BLS predicts strong job growth and high salaries for IT workforce. Retrieved May 2010 from *http://www.cra.org/resources/crn-archive-view-detail/bls_predicts_strong_job_growth_and_high_salaries_for_it_workforce_through_2/.*

[5] [Fincher and Petre 2004] Fincher, S., and M. Petre. 2004. *Computer Science Education Research.* New York: RoutledgeFalmer.

[6] [Forte and Guzdial 2004] Forte, A., and M. Guzdial. 2004. Computers for Communication, Not Calculation: Media As a Motivation and Context for Learning. *Proceedings of the 37th Annual Hawaii International Conference on System Sciences* (*HICSS'04*) 4: 40096.1.

[7] [Green and Petre 1992] Green, T.R.G., and M. Petre. 1992. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings EECE-6* (*6th European Conference on Cognitive Ergonomics*), ed. G.C.v.d. Veer, M.J. Tauber, S. Bagnarola, and M. Antavolits, 167-180. Rome: CUD.

[8] [Green and Petre 1996] Green, T.R.G., and M. Petre. 1996. Usability analysis of visual programming environments: A "cognitive dimensions" framework. *Journal of Visual Languages and Computing* 7(2): 131-174.

[9] [Green et al. 1991] Green, T.R.G., M. Petre, et al. 1991. Comprehensibility of visual and textual programs: A test of "superlativism" against the "match-mismatch" conjecture. In *Empirical Studies of Programmers: Fourth Workshop*, ed. J. Koenemann-Belliveau, T. Moher, and S. Robertson, 121-146. Norwood, NJ, Ablex.

[10] [Guzdial 2003] Guzdial, M. 2003. A media computation course for non-majors. *ACM SIGCSE Bulletin* 35(3): 104-108.

[11] [Guzdial and Ericson 2009a] Guzdial, M., and B. Ericson. 2009. *Introduction to Computing and Programming in Python: A Multimedia Approach*, Second Edition. Upper Saddle River, NJ: Pearson Prentice Hall.

[12] [Guzdial and Ericson 2009b] Guzdial, M., and B. Ericson. 2009. *Problem Solving with Data Structures Using Java: A Multimedia Approach.* Upper Saddle River, NJ: Pearson Prentice Hall.

[13] [Hundhausen et al. 2002] Hundhausen, C.D., S.H. Douglas, et al. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing* 13: 259-290.

[14] [IEEE-CS/ACM 2001] IEEE Computer Society and Association for Computing Machinery, The Joint Task Force on Computing Curricula. 2001. Computing curricula 2001. *Journal of Educational Resources in Computing.* 1(3): 1.

[15] [Johnson and Soloway 1987] Johnson, W.L., and E. Soloway. 1987. PROUST: An automatic debugger for Pascal programs. In *Artificial intelligence and instruction: Applications and methods*, ed. G. Kearsley, 49-67. Boston: Addison-Wesley Longman Publishing Co., Inc.

[16] [Lewandowski et al. 2007] Lewandowski, G., D.J. Bouvier, et al. 2007. Commonsense computing (episode 3): Concurrency and concert tickets. *Proceedings of the third international workshop on computing education research:* 133-144.

[17] [Lister et al. 2004] Lister, R., E.S. Adams, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *Working group reports from ITiCSE on Innovation and technology in computer science education:* 119-150.

[18] [McCracken et al. 2001] McCracken, M., V. Almstrum, et al. 2001. A multi-national, multiinstitutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin.* 33(4): 125-180.

[19] [Miller 1981] Miller, L.A. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal* 29(2): 184-215.

[20] [Moher et al. 1993] Moher, T.G., D.C. Mak, et al. 1993. Comparing the comprehensibility of textual and graphical programs: the case of Petri nets. In *Empirical Studies of Programmers: Fifth Workshop,* ed. C.R. Cook, J.C. Scholtz, and J.C. Spohrer, 137-161. Norwood, NJ, Ablex.

[21] [Naps et al. 2003] Naps, T., S. Cooper, et al. 2003. Evaluating the educational impact of visualization. *Working group reports from ITiCSE on innovation and technology in computer science education:* 124-136.

[22] [Pane et al. 2001] Pane, J.F., B.A. Myers, et al. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies.* 54(2): 237-264.

[23] [Rampell 2010] Rampell, Catherine. "Once a Dynamo, the Tech Sector Is Slow to Hire." *The New York Times*, Sept. 7, 2010.

[24] [Rich et al. 2004] Rich, L., H. Perry, et al. 2004. A CS1 course designed to address interests of women. *Proceedings of the 35th SIGCSE technical symposium on computer science education:* 190-194.

[25] [Simon et al. 2010] Simon, B., P. Kinnunen, et al. 2010. Experience Report: CS1 for Majors with Media Computation. Paper presented at ACM Innovation and Technology in Computer Science Education Conference, June 26-30, in Ankara, Turkey.

[26] [Sloan and Troy 2008] Sloan, R.H., and P. Troy. 2008. CS 0.5: A better approach to introductory computer science for majors. *Proceedings of the 39th SIGCSE technical symposium on computer science education*: 271-275.

[27] [Smith 1975] Smith, D.C. 1975. PYGMALION: A creative programming environment. Computer Science PhD diss., Stanford University.

[28] [Soloway et al. 1983] Soloway, E., J. Bonar, et al. 1983. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM* 26(11): 853-860.

[29] [Spohrer 1992] Spohrer, J.C. 1992. *Marcel: Simulating the novice programmer*. Norwood, NJ: Ablex.

[30] [Tew et al. 2005a] Tew, A.E., C. Fowler, et al. 2005. Tracking an innovation in introductory CS education from a research university to a two-year college. *Proceedings of the 36th SIGCSE technical symposium on computer science education*: 416-420.

[31] [Tew et al. 2005b] Tew, A.E., W.M. McCracken, et al. 2005. Impact of alternative introductory courses on programming concept understanding. *Proceedings of the first international workshop on computing education research*: 25-35.

第8章

超越代码行：我们还需要其他的复杂度指标吗

Israel Herraiz
Ahmed E. Hassan

复杂度存在于软件生命周期的每个环节：需求、分析、设计，当然还有实现。一般来说，开发人员都不希望软件过于复杂，因为这会让软件难于阅读和理解，也就更难修改。此外，高复杂度也被视为造成故障的原因。要想测量复杂度的话，在软件项目的所有产出物之中，源代码最容易入手。不过，在几十年的软件研究之后，人们也没能就如何判断某段代码的复杂度形成共识。甚至对比两段用不同的编程语言编写的代码并指出哪段更复杂也很困难。由于没有一个统一的解决方案，所以现在软件复杂度指标展现出了百花齐放的景象。有没有研究告诉我们如何来根据实际情况选择最佳的指标？这些指标是不是真的比简单的源代码指标（如源代码行数，lines of code，下称LOC）更好？

在这一章中，我们将利用大量开源软件来研究不同的规模和复杂度指标之间的关系。为了避免太多的属性和指标让人看花眼，我们只关注一种编程语言——C，一种软件开发的经典语言，并且直到现在仍然是最流行的编程语言之一。我们对大约30万个文件进行了指标测量，这些指标包括了从最简单和最常见的（LOC）简单指标到非常复杂的语法指标。这样，我们知道了从统计学的角度来看哪些指标是相互独立的，换句话说，就是传统的复杂度指标与简单的LOC指标相比，到底能不能提供更多信息。

8.1　对软件的调查

这个研究的第一步是选择具有代表性的软件样本，就像社会科学中的问卷调查那样。对于已知的对象总体来说，使用统计方法我们将可以从已知对象总体中提取出最小的样本量，而这个样本量可以让我们在一定的不确定性下得出结论。举例来说，在写这篇文章的时候，美国的人口大约是3亿人，那么只要有足够大的样本量（比如3万人），那么我们就可以通过“出口调查”（即在投票站门口对选民进行民意调查以大致估计选举走向）的方式准确的预测整个选举的结果。

要把这种方法运用到软件工程中还面临着一个问题，那就是我们不知道全世界软件的总数

量。这也意味着我们无法确定一个可以在给定不确定性下解决我们的疑问的最小样本量，所以使用传统的“调查方法”来调查所有的软件就不可行了。

虽然我们无法确定世界上有多少软件，但是至少有一部分软件是公开的，这些软件可以用于研究用途，并且开发软件的人们愿意和其他人共享源代码，它们就是开源软件。当然，从理论上讲，如果我们把统计对象总体限制在这类软件中，那么统计的结论就只适用于这类软件。不过，说到底软件开源还是不开源，区别也只是许可的不同而已。虽然开源软件通常都遵循某些实践（如由社区主导开发、开放源代码等），但是其实又各有不同，有的开源项目纯粹由社区主导开发，而有的则仍然受到公司的严密控制。开源软件唯一的共同之处是它们的许可协议，这些协议使得公众可以获取它们的源代码。综上所述，我们可以假定：好的复杂度指标可以来源于开源软件，它的应用范围与开源与否无关。

开源软件还具备了其他一些有利研究的特性。开源软件的储存库对研究人员来说通常是完全公开的。软件储存库中包括了所有软件项目的产出物，包括源代码、版本控制系统、问题跟踪系统以及邮件列表档案等，不但如此，库中还保存了它们的所有历史版本。任何人只要感兴趣都可以访问这些信息。也就是说，开源软件不但提供了大量的代码，让我们研究的样本量大小没有限制，还让我们的研究可复查、可验证，而这些都是实证研究的基本要求，有了它们，研究的结果才值得信赖。

不过，在利用开源软件的过程中，还是有各种危险和陷阱。由于各个开源软件之间的差别较大，所以很难从大量项目（即样本）中获得数据。项目经常使用不同软件库，即便使用了同种软件库用于管理软件库的工具也不尽相同，而这些软件库又常常分散在不同的地点。有这样一些项目，如FLOSSMetrics（http://flossmetrics.org）和FLOSSMole（http://flossmole.org），它们提供了一些目录数据库，其中包括了开源软件的各类指标和信息，这从一定程度上缓解了在对开源软件进行数据挖掘的时候出现的差异性问题。另一方面，开源软件社区自己也用发行版的方式部分解决了这个问题，比如著名的Ubuntu和Debian发行版。发行版从各个开源项目中搜集代码，对其进行修改以便和发行版中的其他软件相互配合，并统一发布为编译好的二进制文件或者源代码包。这些包都被标记上了元信息，这样他们就能对它们进行分类或者下载它们所依赖的其他软件。有的发行版的规模非常大，可能会包括上千个源代码包和上百万行源代码。所以，对于需要收集大量源代码的研究（如本章的研究）来说，这些发行版是研究对象的不二选择。

8.2 计算源代码的指标

我们选择的研究对象是ArchLinux发行版（http://archlinux.org），这个项目包含了上千个开源软件包。ArchLinux是一个轻量级的GNU/Linux发行版，这个发行版的开发人员不愿修改发行版内包含的软件源代码，这样的话，软件包在正式发布之后就可以尽快地被整合到发行版中。在ArchLinux中安装软件包有两种方法：使用官方预编译的包，或者用Arch Build System（ABS）自己从源代码编译安装。

ABS让用户可以获得发行版内的软件的完整源代码。这和其他的发行版不同：其他的发行版

会为软件包的源代码制作副本，并对其打补丁以便能让其融入进发行版的系统中。使用ABS，我们就能自动从软件包的原始位置（如项目网站或者储存库）获取源代码。这就保证了源代码不会被修改，而这也代表样本中的案例研究相互之间没有关联。这样的独立性对于研究结果的有效性来说至关重要，这一点我们将在后面的章节中介绍。

由于ArchLinux的规模较大，用它作为研究对象让我们可以通过ABS获取到上千个开源软件的项目（见示例8-1）。

示例8-1　ArchLinux的编译脚本首部样例

```
pkgname=ppl
pkgver=0.10.2 ❶
pkgrel=2 ❷
pkgdesc="A modern library for convex polyhedra and other numerical abstractions."
arch=('i686' 'x86_64')
url="http://www.cs.unipr.it/ppl"
license=('GPL3')
depends=('gmp>=4.1.3')
options=('!docs' '!libtool')
source=(http://www.cs.unipr.it/ppl/Download/ftp/releases/$pkgver/ppl-$pkgver.tar.gz) ❸
md5sums=('e7dd265afdeaea81f7e87a72b182d875') ❹
```

❶ 软件包的版本。用于生成下载网址。

❷ 次版本发行号。也用于生成下载网址。

❸ 源代码下载网址。

❹ 源代码tarball的校验和。

示例8-1展示了一个ABS编译脚本首部的样例。这个样例包括了各种元信息，这些信息将用于下载源文件、下载其他依赖的软件包以及在软件包编译完成之后对其进行分类。我们使用了在示例8-1中展示的信息来下载所有ArchLinux中的软件包的源代码。

我们使用了SlocCount工具（http://www.dwheeler.com/solccount）来确定下载下来的源代码所使用的编程语言。我们选取了C语言的代码作为分析的样本，并使用Libresoft工具的cmetrics包（http://tools.libresoft.es/cmetrics）计算了多个规模和复杂度的指标。

8.3　指标计算案例

表8-1总结了本次研究中使用的所有指标，以及在后面的图表中代表这些指标的缩写。

表8-1　本次研究中选用的指标

变　　量	指标（对应符号）
大小	源代码行数（SLOC），代码行数（LOC）
	C函数数量（FUNC）
复杂度	McCabe圈复杂度——最大值
	McCabe圈复杂度——平均值
	Halstead长度（HLENG）、体积（HVOLUM）、水平（HLEVE）以及思辨数（HMD）

用于计算这些指标的代码元素如示例8-2所示。这个文件是从urlgfe包（一个跨平台的下载管理器）中提取出来的。（urlgfe最近将名字改为了uget，所以现在你将无法在ArchLinux中找到urlgfe。）这个文件包含了预处理指令（如❶）、注释（如❸），只有一个函数（从❹开始），这个函数包含了一个while循环。

示例8-2　C源文件示例

```
#ifdef HAVE_CONFIG_H ❶
# include <config.h>
#endif
❷
/* Specification. */ ❸
#include "hash-string.h"

/* Defines the so called `hashpjw' function by P.J. Weinberger
   [see Aho/Sethi/Ullman, COMPILERS: Principles, Techniques and Tools,
   1986, 1987 Bell Telephone Laboratories, Inc.] */
unsigned long int ❹
__hash_string (const char *str_param)
{
  unsigned long int hval, g;
  const char *str = str_param;

  /* Compute the hash value for the given string. */
  hval = 0;
  while (*str != '\0')
    {
      hval <<= 4;
      hval += (unsigned char) *str++;
      g = hval & ((unsigned long int) 0xf << (HASHWORDBITS - 4));
      if (g != 0)
        {
          hval ^= g >> (HASHWORDBITS - 8);
      hval ^= g;
        }
    }
  return hval;
}
```

❶ 预处理指令，算入LOC和SLOC。

❷ 空行。算入LOC，但是不算入SLOC。

❸ 注释行数。算入LOC，但是不算入SLOC。

❹ 代码，算入LOC和SLOC。

我们可以计算的最简单的指标是和代码行数相关的指标（如LOC和SLOC）。函数的数量也很容易就能从源代码中计算出来。其余的复杂度指标，即McCabe的圈复杂度和Halstead的软件科学指标系列，则稍微要复杂一些。

除了McCabe的圈复杂度以外的指标都是按文件来定义的。所以，我们为所有C文件（由SlocCount识别）计算了这些指标，并忽略了头文件（即名字结尾为.h的文件）。

而根据定义，对McCabe的圈复杂度的计算必须是基于整个函数或者程序，因为这个指标的

对象必须包含一个起点和一个（或多个）终点。我们决定计算每个函数的圈复杂度，并将一个文件的圈复杂度按两个方法做总结：一个是该文件的所有函数中复杂度中最高值，一个是平均值。

此外，我们还为每个文件都计算了MD5散列码，这样我们就可以把重复的文件从统计分析中剔除，因为在统计中包含多个同样的文件会对结果造成影响。

8.3.1 源代码行数（SLOC）

我们采用了Conte对于这个经典指标的定义[1]：

> 代码行是指除了注释和空行之外的所有程序文本行，无论这一行中包含了多少条语句或者语句段。这个定义具体包括所有的程序头首（header）、各种声明以及可执行和不可执行的语句。

以示例8-2中的文件来说，如果我们忽略空行和注释，但是把预处理器指令和剩下的代码行算进来的话，这个文件就包括23 SLOC。

8.3.2 代码行数（LOC）

我们使用Unix的wc工具来计算了每个源文件的总代码行数，这次包括了注释、空行等。

这个指标非常简单直白，因为它算上了所有的空行和注释，等等。示例8-2包括了32 LOC（另有18 LOC是许可证的注释文字，与本次研究无关所以删除了）。

8.3.3 C函数的数量

我们结合使用了exuberant-ctags和wc工具来计算每个文件内的函数数量。

这个指标更好计算。示例8-2中只包含了一个函数，所以这个文件的CFUNC是1。

8.3.4 McCabe圈复杂度

对于这个指标，我们采用的是McCabe原论文中的定义[6]，即如何把源代码文件用图和区域来表示。任何程序都可以表示为一个图。如果我们用图来表示一个最简单的语句，即没有条件、循环、分支的语句，就会是图8-1中(a)图的样子。`if`语句会被表示为一个分叉，如图8-1中(b)图的样子。循环语句则会被表示为一条边，这条边将返回上一个节点。

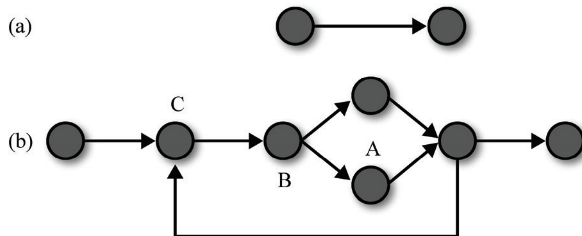


图8-1 两个图示例。图(a)的CYCLO是1，而图(b)的CYCLO是3

如果一个图G有n个顶点，e条边和p个出口点（如，对于C语言来说就是函数返回），其复杂度v的定义如下：

$$v(G) = e - n + 2p$$

圈复杂度指标（CYCLO）的起始值是1，代表一系列按顺序执行的无分叉或者循环的语句。每在控制流程图中增加一个区域，CYCLO的值就增加1。比如说，如果一个程序包含了一个`if`语句（没有`else`）的CYCLO值就是2，因为这个语句在控制流程图中额外生成了一个新的区域。

示例8-2中的程序的控制流程图如图8-2b所示，其CYCLO值是3，其中`if`分叉生成了一个新的区域（即图中标出的A区域），而`while`循环则生成了另一个区域（即图中标出的B区域）。还有一个C，代表周边区域，这个区域在圈复杂度中永远算作1。

8.3.5 Halstead软件科学指标

在这里我们使用了Kan对这个指标的定义[5]。需要计算的有4个指标：长度、体积、水平和思辨数。这些指标基于程序中的操作符和操作数的冗余程度。

在C语言中，操作数是字符串常量和变量的名字。操作符则包括各种符号（如+、-、++和--）、间接操作符`*`、`sizeof`操作符、预处理器常量、控制语句（如`if`和`while`）、储存类指示符（如`static`和`extern`）、类型指示符（如`int`和`char`）以及结构指示符（如`struct`和`union`）。

要计算指标，首先需要计算程序使用了多少个不同计算操作符（n_1），多少个不同的操作数（n_2），操作符的总数（N_1）以及操作数的总数（N_2）。程序的长度L是操作符与操作数的数量之和：

8

$$L = N_1 + N_2$$

程序的体积V是其实际大小，定义为：

$$V = N \cdot \log_2(n_1 + n_2)$$

程序的等级lv的上限是1，等级越接近1则程序越简单。等级定义为程序的体积和其潜在体积（即最少冗余的算法实现）的比率：

$$lv = \frac{2}{n_1}\frac{n_2}{N_2}$$

把这个比率的两方倒转，就可以得到常常被人称之为“代码难度”的指标。最低的难度是1，上不封顶。

程序员需要花在理解程序上的工作量E定义为：

$$E = \frac{V}{lv}$$

这个指标有时也被称之为程序员为了理解一个程序所必须的*思辨*（mental discrimination）*次数*。

Halstead把编程语言和自然语言之间做了一个类比，并得出了这些公式。其中心思想是，算法的实现是用语言来表述的，如果这个表述较短或者包含高冗余度的操作符及操作数的话，要理解起来就更容易，因为这样程序员需要同时放在记忆中的概念数量就会少一些。举例来说，

Halstead的等级和操作数的冗余度有关。如果冗余度很高，有很多重复的操作数，Halstead的值就会较低，这表示程序复杂度较低。这种方法很好理解，因为这样的冗余可以让人更快地学习和理解程序。

Halstead软件科学指标和McCabe的圈复杂度类似，都是针对非模块化的程序整体来定义的，不包括这个程序导入的其他元素。和McCabe的指标不同的是，Halstead的指标并不依靠代码的结构来计算复杂度。也就是说，Halstead的指标纯粹是基于组成程序的文本元素来定义的，而不是基于语法单位（如函数），所以我们把整个文件作为了Halstead指标的计算对象。

示例8-2中所示的样例文件的Halstead长度为97，即操作符（字符串常量和变量名）加上操作数（符号、语句等）之和。Halstead体积是526，Halstead等级是0.036。思辨次数是14490，即体积和等级的商。几乎所有的指标都是数值越大越复杂，除了Halstead等级，这个指标是数值越小越复杂。

8.4　统计分析

ArchLinux库中包含了4096个软件包（截至2010年4月），其中有些包是同一个软件的不同版本。在移除了不同的版本之后，我们得到了4015个软件包，共1272748个源代码文件。在这些文件中，有576511个文件是用C语言写的。不过，其中还有重复的文件。最后，我们共找出776573个互不相同的文件，其中用C语言写成的只有338831个。在这些C语言源文件中，有212167个不是头文件，剩余126664个是头文件。

我们按照前一节中描述的针对示例8-2中的计算方法为所有这些C语言源文件进行了指标的计算。最后，我们得到了超过300000条指标数据。每条数据都包含了一组9个指标的元组。

8.4.1　总体分析

我们首先对这9个指标之间的相互联系在文件级别上进行基本分析。目标是对这9个指标进行标准化的工作，以得到一套统一的软件规模和复杂度指标。我们的目标是从这些指标中找出无法提供进一步信息的，并移除。

在分析中，我们把每个文件都看做是一个独立的数据点。这个假设是比较适当的，因为重复的文件已被移除，而且这些文件都来自相互独立的项目。为了研究相关性我们决定使用指标值的对数值。我们是在测算了最佳理想的Box-Cox权重变形函数之后做出了这个选择。在得到了所有指标的对数值之后，我们又对它们进行了线性回归分析，并计算了他们的Pearson相关系数。Pearson系数接近1就代表两个变量是紧密相关的，而如果系数接近0，则意味着变量之间互不影响。[①]

表8-2展示了所有的指标之间的Pearson相关系数。这个表格是对称的，也就是说，在对角线上下的值是一样的，因为无论谁先谁后或者谁依赖谁，两个变量之间的相关系数都是相同的。所以在表8-2中我们只保留了一半的数据，以免让人感觉混乱。大部分指标之间的相关系数都非常

① 维基百科包含了关联性分析和Box-Cox权重变形函数的详细信息，详见 http://en.wikipedia.org/wiki/Correlation_coefficient 和 http://en.wikipedia.org/wiki/Box-Cox_transformation（于2010年4月3日访问）。

高，这表明了他们之间有非常密切的关联。我们着重展示了LOC、SLOC同其他指标之间那些较高的相关系数，由此我们可以看出大部分的复杂度指标其实都和这两个简单的指标相互呼应。高相关性意味着两个指标所提供的复杂度的信息量差不多。比如，Halstead的长度以及SLOC的关联系数高达0.97，所以我们用它们中的任何一个来确定文件的复杂度结果都不会有什么差别。

表8-2 指标之间的关联系数

	SLOC	LOC	NFUNC	MCYCLO	ACYCLO	HLENG	HVOLU	HLEVE	HMD
SLOC	1.00	**0.97**	0.68	**0.77**	0.63	**0.97**	**0.97**	0.88	**0.96**
LOC		1.00	0.67	**0.75**	0.60	**0.94**	**0.94**	0.84	**0.92**
NFUNC			1.00	0.63	0.32	0.64	0.63	0.67	0.66
MCYCLO				1.00	0.91	0.76	0.75	0.82	0.80
ACYCLO					1.00	0.63	0.62	0.72	0.68
HLENG						1.00	0.99	0.90	0.99
HVOLU							1.00	0.90	0.98
HLEVE								1.00	0.96
HMD									1.00

有的复杂度指标则和其他的不太相关。比如，McCabe复杂度的最大值（即表8-2中的MCYCLO）和平均值（ACYCLO），与SLOC/LOC的关联度都不高。我们能否找到原因呢？

MCYCLO和ACYCLO都和代码的结构有关：分叉和循环越多，这些值就越高。这可能造成了McCabe指标和代码行指标之间的不统一，即头文件和非头文件之间的区别。

两类文件我们都做了分析。在C语言中，头文件大多包含的是函数和实体的规范信息。不过，它们还可能包括条件预处理指令，这些指令都是编译时解析的，而不是在运行时。虽然条件预处理指令也属于分叉的一种，但是我们的工具并不把它们看做是程序控制流程图中的区域。所以，无论头文件是大是小，它们的McCabe圈复杂度都会偏低。也就是说，如果想要找出代码行和圈复杂度之间到底有没有关系，我们应该把头文件也从统计中移除。

8.4.2 头文件和非头文件之间的区别

如果我们把这些样本分成两份，一个对应头文件而另一个则对应非头文件，相关系数的值就会改变。

我们这里只包含了这些指标中的几个。这里没有包含的指标在原来的总体分析和后面把样本分成两份的分析中，相关系数都非常高。所以，为了简明起见，我们在结果的表格中省去了这些指标。

表8-3列出了头文件的统计结果。我们着重显示了两个相关系数，这两个系数表明在头文件

中，圈复杂度和SLOC之间的关联度很低。另一个复杂度系数，Halstead等级（即表中的HLEVE）则和SLOC密切相关。这个结果在意料之中，因为头文件无论有多大，都是属于没有分叉一路到底的类型。所以，这些文件的圈复杂度总是会很低，所以和文件大小（即LOC/SLOC）的相关性就会很小了。比如说，圈复杂度最大值（MCYCLO）在全部文件中的平均值是2，中位数是1，这表明大部分的文件的MCYCLO都小于等于2。但是，由于圈复杂度和文件大小之间相互独立，这意味着我们不能使用圈复杂度作为衡量头文件复杂度的指标。此外，有趣的是，所有其他的复杂度指标都和文件大小紧密相关。

表8-3　头文件的关联系数

	SLOC	MCYCLO	ACYCLO	HLEVE
SLOC	1.00	**0.37**	**0.34**	0.72
MCYCLO		1.00	0.97	0.49
ACYCLO			1.00	0.46
HLEVE				1.00

表8-4展示的是非头文件的统计结果。这次，所有的相关系数都很高。只有一个与文件大小的相关性不太高，那就是ACYCLO，圈复杂度的平均值。但是，这个指标和圈复杂度最大值（MCYCLO）的关联系数很高，而MCYCLO本身和文件大小是紧密相关的。Halstead等级（HLEVE）也和非头文件的大小紧密相关。

表8-4　非头文件的关联系数

	SLOC	MCYCLO	ACYCLO	HLEVE
SLOC	1.00	0.83	0.60	0.91
MCYCLO		1.00	0.86	0.82
ACYCLO			1.00	0.65
HLEVE				1.00

那么，从这些紧密关联的指标中，我们又能得出什么结论呢？是不是说简单的代码行和那些复杂的指标（比如McCabe和Halstead）一样好？在开始总结之前，让我们先看看一些软件指标的相关文章。

8.4.3　干扰效应：文件大小对相关性的影响

在研究界，用源代码作为样本来对各种指标进行实际验证非常流行。这些研究中的大部分都和本章介绍的研究相似。El Eman等人[2]对很多验证研究进行了一次详细的分析，并就这类统计研究的有效性提出了一些质疑。尤其是，研究者们证明了类的大小会影响到用于预测故障的面向

对象的指标的有效性。如果重复同样的实验，只是对实验对象的大小加以控制，那么很多结论就无法成立。

由于我们的研究的对象是文件而不是类，而且不同类别的指标的设计理念又不同，所以我们无法在这次研究中直接应用这种方法，不过我们仍然应该考虑到我们的指标有可能会被文件大小所干扰。为了测试是否存在干扰效应，我们把所有的结果按文件大小分类，看看相关系数会不会随着文件的大小而改变，也看看我们的结论会不会随着文件大小而改变。

1. 文件大小对于头文件的相关性的影响

前面我们已经介绍过，McCabe的圈复杂度和文件大小的相关性很低，然后我们认为不应该为头文件计算这个指标，因为头文件和分叉的关系很不清楚。图8-2展示了在文件大小变化的时候，圈复杂度（竖列）和SLOC（横列）之间的相关系数的变化。我们把文件分成了20个等级，每个包含了同样数量的文件，并只用这些文件来计算相关系数。由于文件大小的范围很广，横列用对数刻度展示。两条曲线分别代表圈复杂度最大值（MCYCLO）和圈复杂度平均值（ACYCLO）。

乍一看，对于很小的文件来说，二者几乎没有什么关联（关联系数接近0），而关联系数随着文件大小增长，直到差不多 500 SLOC的时候，才平稳了下来。对于非常大的文件来说，相关系数的变化不大。

我们可以从图8-2中得出两个结论。首先，非常大的文件应该从样本中移除。不过，这些文件对于相关系数的影响非常小，这一点我们可以从图中的相关位置很少的增长中看出。第二个结论是文件的大小确实会对相关性产生影响。不过这不会对我们的头文件统计结果造成影响，因为无论文件多大多小，相关系数都非常低。不过，其余的相关系数又会受到怎样的影响呢？

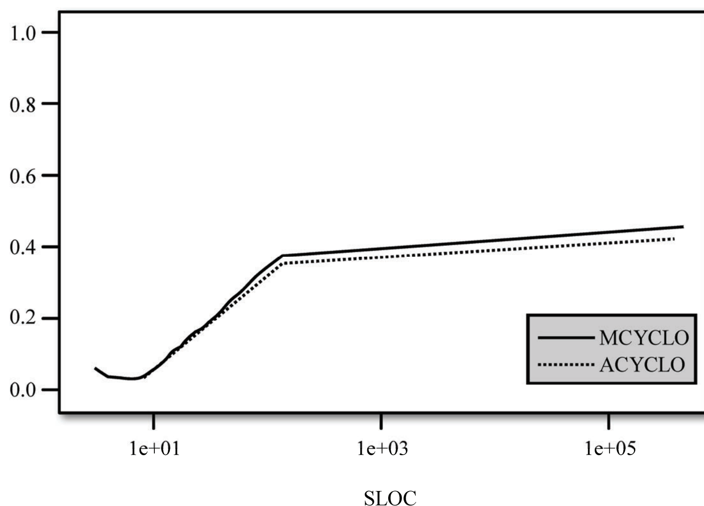


图8-2　头文件的大小对于其圈复杂度和SLOC的相关性的影响

2. 文件大小对于非头文件的相关性的影响

图8-3展示了非头文件的大小对于其圈复杂度和SLOC的相关性的影响。竖列展示的是相关系

数的值，而横列仍用对数刻度展示，表示文件的SLOC值。两条曲线分别代表圈复杂度最大值（MCYCLO）和圈复杂度平均值（ACYCLO）。

和上次一样，我们可以观察到文件大小对于相关系数确实有影响。不过，这些值在文件达到一定大小之后就稳定了（对MCYCLO来说约为1000SLOC，对ACYCLO来说约500SLOC）。对于非常小的文件来说，低相关系数是可以理解的，因为小文件的差异度非常之高。对于中等大小的文件来说，相关系数和前一节中展示的类似。

3. 文件大小对Halstead软件科学指标的影响

图8-3展示了文件的大小对于Halstead指标和SLOC的相关性的影响。和之前相同，竖列展示的是相关系数，横列按对数刻度展示文件大小，4条曲线每条代表一个指标。

这里，曲线的模式和前面的差不多，尽管所有指标的一致性更高一些。虽然文件大小对相关系数造成了影响，但是无论文件多大多小，相关系数都很高。

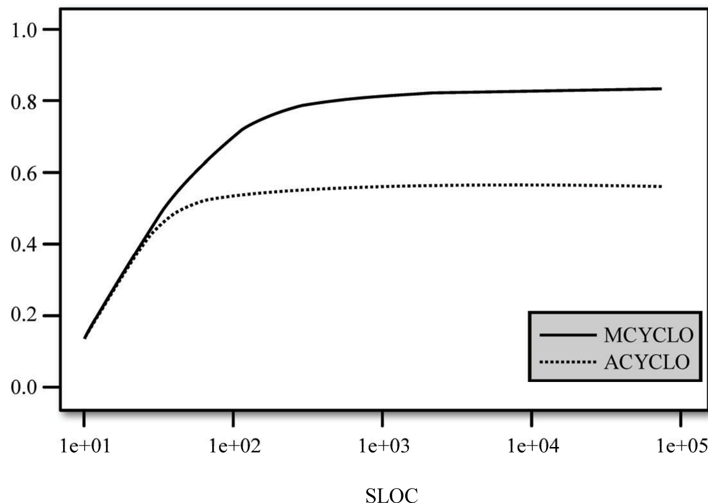


图8-3　非头文件的大小对于其圈复杂和SLOC的相关性的影响

4. 关于文件大小的干扰效应的总结

虽然文件的大小的确会影响相关系数的值，但是这样的影响并不会对我们从相关系数中得出的结论造成很大影响。事实上，类的大小对于面向对象的指标的干扰效应的观点也受到了一些批评[4]。

在本次研究中，我们可能应该移除非常大的文件，因为这些文件也许会使我们的结论产生偏差。我们对这些文件进行了随机抽查。有的文件似乎是自动生成的，所以不应该出现在源代码复杂度的分析中。也就是说，这些文件不会由程序员来阅读和修改，因为它们都是自动从程序员写的源代码文件中自动生成出来的。不过无论如何，这样的影响也不会很大，正如我们前面的几张图所示。

我们这里提供一些实用的建议，当在其他项目中应用本章所展示的方法的时候，我们推荐研究员们按文件大小来进行分析，这样就可以测试是否相关系数会随着文件大小而改变。

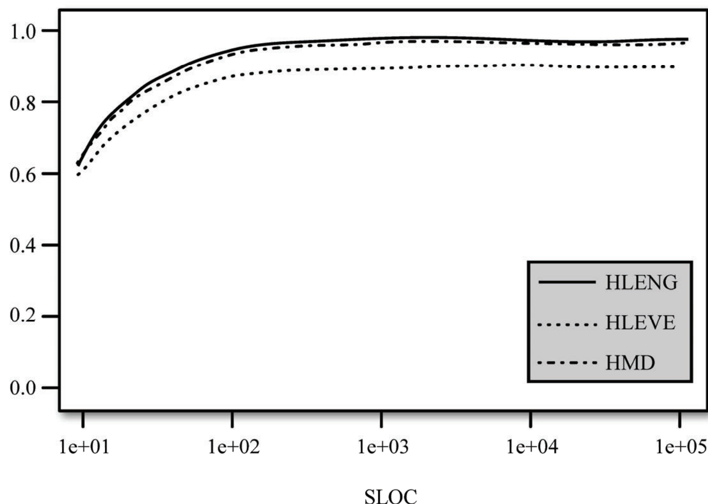


图8-4　非头文件的大小对于其Halstead软件科学指标和SLOC的相关性的影响

8.5　关于统计学方法的一些说明

统计学的分析会从数据中提取结论，而有很多因素都会威胁到这些结论的有效性。回到本章开头“出口调查”的例子，受访者在社会经济地位上的不同可能会影响出口调查的结构。要想做一份可靠的出口调查，必须随机选择受访者，否则的话，调查就无法准确地预测选举的实际结果。同样，本章所展示的统计结果可能会受到别的什么因素的干扰，而我们的结论也可能无法适用于所有的软件项目。为了我们的研究的完整起见，我们这里讨论一下会对研究有效性有威胁的因素。

- 你可能首先会注意到本章所示的相关系数的统计显著性问题。由于统计学上的特性以及样本量的规模巨大(达到了十万级)，本次研究数据的显著水平一直都非常高(达到99.99%级)。
- 从软件开发的角度来说，这次研究应该扩展到其他的编程语言。这里所展示的C语言相关结论可能对其他语言并不适用。虽然C是目前开源社区中最流行的编程语言（按现有可利用的代码来算），但是还有其他很流行的编程语言，它们增长的速度比C快很多，而且也有大量的代码可以利用。
- 最后本次研究所涉及的所有源代码都是从公开的开源项目中得来。虽然我们相信开源的软件和其他类型的软件之间并不存在技术上的差异，但是这种对样本的选择还是可能会影响到我们结论的有效性。尤其是，我们的结论可能不适用于在不同的条件下开发的软件。要想解决这个有效性的问题，就应该使用其他领域（如商业、科学软件）中得来的代码来重复本次实验。

8.6　还需要其他的复杂度指标吗

本章所展示的结果表明，对于用C语言写成的非头文件来说，所有的复杂度指标都和代码行

紧密相关，所以更复杂的指标并不能提供比简单的代码行更多的信息。

不过，在接受这个结论的时候，有几点需要注意。对头文件的分析显示，圈复杂度和其他指标的相关性很低。我们认为这是由这类文件的性质决定的。换句话说，头文件中没有实现，只有规范信息。我们想要用“理解程序”这个概念来定义和计算复杂度。当然，程序员也需要阅读和理解头文件，这也意味着头文件也会造成一定程度的复杂度。不过，圈复杂度和代码行之间的相关性很低并不意味着这个指标就是一个好的头文件复杂度指标。正相反，这样的低相关性只是由于头文件缺乏控制结构。这些文件中没有循环、分叉等，所以无论文件大还是小，它们的圈复杂度都最低。

对于非头文件来说，所有的指标都显示出了与LOC非常高的相关性。我们测试了文件大小的干扰效应，结果显示，无论文件大小，相关性都很高。

我们认为，这次研究明确地教会了我们一课，即语法上的复杂度无法包揽软件复杂度的全部。仅仅基于程序结构或者文字特性（比如Halstead指标所关注的冗余度）的复杂度指标无法告诉我们理解一段代码需要的工作量，或者我们至少可以说，这些复杂的指标并不能比代码行告诉我们的更多。这会对我们如何应用这些指标造成影响。特别是缺陷预测、开发和维护工作量模型以及统计学模型，通常来说，它们无法从这些指标上获得更多的好处，而代码行应该被视作这些模型的首选和唯一指标。

软件研究界曾经面临过代码复杂度和理解复杂度的对比问题。现在已经有人提出了一种语义熵指标，这种指标基于程序中标识符的含糊程度。有趣的是，这种指标非常适合于故障的预测[3]。

这并不意味着我们没有从传统的复杂度指标身上学到什么。首先，圈复杂度非常好的表示出了测试程序时所需遍历的路径数量。Halstead的软件科学指标也给了我们一些有趣的经验：总是有很多不同的方法来做同一件事。所以，如果你选择一种方式并在程序的多个地方进行运用，你会让你的代码的冗余度增高，这会让代码更简单、更容易阅读——即便其他的统计数据不这样认为。

8.7 参考文献

[1] [Conte 1986] Conte, S.D. 1986. *Software Engineering Metrics and Models*. San Francisco: Benjamin Cummings.

[2] [El Emam et al. 2001] El Emam, K., S. Benlarbi, N. Goel, and S. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27(7): 630-650.

[3] [Etzkorn et al. 2002] Etzkorn, L.H., S. Gholston, and W.E. Hughes, Jr. 2002. A semantic entropy metric. *Journal of Software Maintenance and Evolution: Research and Practice* 14(4): 293-310.

[4] [Evanco 2003] Evanco, W. 2003. Comments on “The confounding effect of class size on the validity of object-oriented metrics.” *IEEE Transactions on Software Engineering* 29(7): 670-672.

[5] [Kan 2003] Kan, S.H. 2003. *Metrics and Models in Software Quality Engineering* (2nd Edition). Boston: Addison-Wesley Professional.

[6] [McCabe 1976] McCabe, T.J. 1976. A complexity measure. *IEEE Transactions on Software Engineering* SE-2(4): 308-320.

Part 2

第二部分

软件工程的特有话题

这一部分把之前那部分的想法运用于编程和软件开发领域的若干迫切话题中。这些话题中的大部分是有普遍争议的，呈现多方争鸣的态势，每一方都有热情的支持者。程序员对这些问题的态度会影响他们的日常行为，并控制雇佣和预算的决定。你会在这一部分的书中学到我们能在多大程度上相信当前这些信念。

第9章

自动故障预报系统实例一则

Elaine J. Weyuker
Thomas J. Ostrand

每个人都希望自己开发的软件完美无缺，没有故障（或者缺陷、bug）。通常，开发人员会使用软件测试技术来找出这些问题然后解决，但是当软件系统的规模很大的时候，全面测试可能会耗费大量的人力物力。

如果可以准确预测出系统中最有可能出现故障的部分，那么开发人员、测试人员以及管理者们就可以大量节省寻找这些故障的时间，如此，不但能提高测试效率，也能提高短缺资源的利用率。在过去的几年里，我们设计了一些数学模型，用于预测软件系统在下一个版本发布时每个文件中的故障数量。然后，开发人员就可以按照预估故障数量由多到少来为这些文件排序，所以很容易看出哪些文件最可能出故障。

现在，我们用其中一个模型做了一个自动化的工具，用于帮助项目管理者优先分配工作给最容易出故障的文件。例如，项目经理可以为这些文件安排更多的测试时间，指派团队中最好的开发人员来修改这些文件，或者为这些文件进行代码检查。这样，就可以更快地发现故障，并降低测试的成本。此外，这样的方法还可能会空出一些资源，使项目组可以对整个系统进行更多的测试，这样软件的可靠度就会更高。

9.1 故障的分布

大型的企业级软件系统通常开发周期很长，参与的开发人员也很多，而且会定期发布更新版本。一般坊间的看法是，大型系统经过几个早期版本之后，在后续版本中的bug通常会集中在相对较小的一个部分的代码中。这样的bug分布常常被称之为*帕累托分布*（Pareto distribution），即80%的bug都存在于20%的代码实体（如文件或方法）中。如果系统测试和调试人员可以找出哪些文件属于那20%会出问题的文件的话，他们的工作就会轻松很多，因为这样他们就能高效地专注于质量保证的工作（比如测试、检测和调试），而不是在寻找问题上浪费时间。

在AT&T工作期间，我们接触到了不少开发周期非常长的大型系统，所以做了一份研究。研究目的如下：

- 在系统测试阶段之前识别出代码中最可能出现bug的部分；
- 设计和实现一个编程环境工具，用于识别系统中最容易出现bug的部分，并把这个信息展示给开发人员以及测试人员。

当然，只有故障真正集中在系统的某些部分，才会有“系统中最容易出现bug的部分”这个概念。所以为了确保这个概念的可行性，我们首先需要找到证据来证明bug在代码中的分布确实是像帕累托分布那样高度倾斜的。

我们研究了AT&T不同部门经常使用的6个大型系统，结果如表9-1所示。它们的目的包括库存控制、供应调度、维护支持、自动语音应答处理。这些都是真实的商业系统，它们产生的bug也都是真实存在过的，大部分这些bug都在内部测试中被发现并处理了，但是也有一些在发布之后才被用户发现。

有三个系统是由内部的软件开发组织来开发和维护的，而另外三个则是外包公司做的。他们的规模从30万行到超过50万行源代码不等，而生命周期则是从2年到接近10年。每个系统都包括了不同编程语言的代码，少的有4个，而有一个系统则超过50种语言。这些系统的开发历史严格遵循定期发布版本中的规律，通常每三个月发布一个新版本，除了一个例外（后面将会讲到）。

从对这些系统的研究中得到的数据完全支持关于bug的帕累托分布假说。在对第一个系统进行的预备研究中，我们发现在其第2到第9个版本时，bug集中在少于20%的文件中，而在第9个版本之后的版本中bug则更为集中地存在于少于10%的文件中。

表9-1 系统总览

系统	总版本数	正式发布使用时间（年）	平均每版本文件数	平均每版本KSLOC	平均每版本的bug数	平均每版本包含bug的文件%	文件中包含bug的代码行数%
库存管理	17	4	1318	363	301	10.4	28.7
供应调度	9	2	2178	416	34	1.3	5.7
语音应答	9	2.25	1341	228	165	10.1	16.9
维护支持A	35	9	550	333	44	4.8	17.4
维护支持B	35	9	1237	300	36	1.9	13.7
维护支持C	27	7	437	219	42	4.8	14.3

这使我们确信，设计故障预测模型确实是有意义的。在对这些系统进行的研究中，我们不断发现新的证据来证实帕累托分布确实存在，也就是说，这些项目中的故障都令人吃惊地集中在代码的非常小的部分里。其中有两个系统，平均每个版本bug文件的百分比少于10.5。还有两个系统的bug只存在于5%的文件中。剩下还有两个系统，只有2%的文件中包含了bug。说得更明白点，就是：在我们所研究的6个系统中，大概有90%（甚至更多）的文件在发布前的测试以及正式发布以后的使用中都没有发现bug。虽然包含bug的文件常常会比不包含bug的文件要大一些，但是6个系统中有5个，其中有bug的源代码行数也只占到不到总量的18%而已。

表9-1展示了每个系统的bug集中度，以及其他的一些重要的数据。表的第2和第3列用发布的版本数量以及存在的时间长度来表示这个系统的年龄。剩余的几列所展示的是每个系统全版本的平均值。这些数值分别是：文件数量、有多少千行源代码（KSLOC）、平均bug数、包含bug的文

件占文件总数的百分比，以及包含bug的代码行数占总行数的百分比，这些数值体现了系统的规模。

一般说来，系统开发周期越长，新增加的功能也就越多，文件的数量也会随着增长。所以，表9-1中所示的平均文件数量以及平均代码行数比那些较晚发布的系统要少一些。表9-2展示的是每个系统最新版本的文件数量以及源代码行数，以此来判断其规模大小，此外还有我们的预测结果。

当开发组织非常大，参与开发的程序员和测试人员非常多的时候，就必须要有系统的缺陷报告机制、变更管理以及版本控制。我们研究的这些系统使用了同一个工具来解决这些问题。这个工具会把所有对代码的修改记录在一个*修改请求*（Modification Request，下简称MR）中，这个MR将描述需要的修改，并记录下实际做了的修改。MR可以请求对系统任何方面的修改，包括系统需求到设计文档、代码、使用文档等，都可以。

表9-2　这些系统包含bug最多的20%文件中包含的故障百分比

系　　统	最新版文件数	最新版KSLOC	20%最多bug的文件中包含了故障%
库存管理	1950	538	83%
供应调度	2308	438	83%
语音应答	1888	329	75%
维护支持A	668	442	81%
维护支持B	1413	384	93%
维护支持C	584	329	76%

系统设计师想要改变某些功能可以提交MR，程序员想要为他负责的程序的一部分采纳更有效的算法的时候可以提交MR，而测试人员在发现程序未能正常运行的时候也可以提交MR。如果是程序员提交的MR，那么在提交之后他可能马上就可以进行修改。也有可能MR提交之后会经过数小时甚至数天的时间才正式开始修改，而且还有可能不直接由提交者来修改，当提交人是测试人员的时候尤是如此。MR将记录关于修改的一切信息，包括MR在提交和实施时项目所处的阶段以及日期、提交人以及实施者的身份、用于描述MR的相关标签信息以及提交人和实施者对这个修改的文字描述。

MR还将记录实际的修改，这些信息将由这个工具的版本控制组件记录，以便可以将系统恢复到开发生命周期的任意时间点上。所有这些MR信息都储存在数据库中，我们的故障预测工具只需要利用这些信息来分析过去的版本就能预测未来版本中可能出现的故障。

虽然我们研究的系统都使用了同一个工具来记录MR，但是每个系统的使用方式又可能有所不同。最大的不同在于项目从哪个开发阶段开始要求所有的修改都必须通过MR的记录。通常的做法是在进入系统测试阶段之后才要求用MR来记录修改，也就是说MR数据库不记录系统测试之前产生的修改或者发现的问题。我们研究的6个系统中有4个采用了这样的做法。对这4个系统来说，我们希望我们的模型可以预测系统测试以及正式发布使用阶段的故障，因为模型是基于这些阶段以往的错误建立的。

此外，库存管理系统记录了从单元测试开始的所有开发阶段（包括后面的系统测试以及正式发布使用）中发现和修正的故障。这个系统的故障约有80%是在单元测试和集成测试阶段发现的

（即在系统测试阶段之前发现的），所以其平均故障数量相比其他系统要高。这就解释了为什么库存管理系统的故障看上去要比其他任何系统都要多很多。数量多并不一定就表明这个系统的问题比其他系统更多。

语音应答系统是一个特例，我们将在下一节进行讨论。

9.2 故障高发文件的特征

在证明故障具有集中性之后，我们就开始分析这些频频出现故障的代码和文件到底有些什么特征，以便以后可以更好地进行识别。处在开发阶段的软件有两种属性可以帮助我们分析代码。第一种是可以直接从源代码中提取出来的*静态代码结构属性*。这些属性包括编程语言、文件的代码行数、方法调用次数以及各种复杂度指标。

第二种是*流程相关的属性*，比如和系统的开发测试过程相关的一些属性。比如：前个版本中发现的修改以及故障数量、某个部分的代码在系统中存在的时间等。

我们早期研究的目标是分析故障文件的这两种属性，并从中找出和故障的发生密切相关的属性和特征。我们研究的前两个系统提供了足够的证据，使我们可以构建一些初级模型。而这些模型都较为成功地预测了哪些文件在这两个系统未来版本中最易发生故障。

我们研究的第三个系统，即语音应答系统，虽然和前两个系统在规模、正式发布使用的时间以及开发人员数量上都差不多，但是其使用的开发流程阶段的划分和发布时间间隔却和常见的不同。我们无法对这个系统的故障及其开发阶段做关联。

这个语音应答系统的发布是不间断的，任何时候只要修复了bug或者添加了新功能，就会发布一个版本。由于这个系统没有定期发布新版本的制度，而且也没有相应充分的系统测试，我们也不清楚基于前两个系统的模型是否也适用于这个系统，但是我们却惊喜地发现虽然稍有差异，但是这些模型的表现都非常好。表9-1和表9-2中展示的所谓语音应答系统的各个“版本”实际上只是截取了这个系统的生命周期中连续的三个月里所发布的版本，而相关的bug也是指在这三个月内首次发现的bug。开发模式上的不同可能是造成这个系统每个版本的故障比其余大多数项目都要多的原因。这个项目的很多MR都是记录开发人员在模块开发较早时期所做的修改，这对应了传统项目的单元测试阶段。

基于我们对前三个系统的实验，我们确定了一个标准的预测模型，并对三个维护支持系统进行了预测，结果极好。

9.3 预测模型概览

要想预测某个系统在第N个版本中的故障，我们就必须分析早前版本中的代码特质、流程特质以及故障数量，并生成相应的模型。这个方法的基本前提假设是：在早前版本中和故障文件紧密相关的特质，在后续的版本中也会保持这种相关性。预测模型由一个方程式组成，而其中的变量则是版本N中的文件的代码和流程特质，得出的结果则是这个文件的预测故障数量。要创建模型最少必须要有两个早前版本的数据，上不封顶。用于建立模型的数据我们称之为*养成数据*

（training data）。

因为可以得到这些系统的所有历史故障数据，所以可以利用从版本1到版本N－1的信息来生成版本N的预测模型，然后再查看版本N的实际故障分布来检查预测结果的准确度。我们评估预测结果的方法是检查排在预测列表前20%的文件中包含的故障数量。表9-2显示，前20%的文件中至少包含了75%的故障，大部分时候这个比率甚至超过了80%。这个表有力地证明了这个预测方法也适用于其他的大型项目。

你可能会好奇，为什么会选择一个看似随机的比率20%来评估预测结果呢？原因有两个。首先，我们和其他研究人员的研究都表明，绝大部分的故障都存在于故障最多的那20%的文件中，这使我们认为：如果预测出的文件包含了80%的故障，那么“找出故障频发的文件”这个任务就算圆满完成了。

其次，文件总数的20%通常不会太多，这样我们想要重点关注故障文件的想法才能得以实施。当然，并不是说其他的文件就不值得仔细测试或者检查了，只是说那20%的文件可能会需要额外的关注。

我们使用的这些预测模型都基于负二项式回归分析（Negative Binomial Regression，下称NBR）[2]，因为这是我们找到的针对这次研究最有效的统计方法。我们建立了NBR模型的一种形式，并成功地将其应用到了很多不同的系统中。NBR是一种广为人知的统计建模方法，也是线性回归的扩展。NBR尤其适用于计算结果是非负整数的情况，如预测软件系统中某个文件的故障数量。NBR的模型将计算结果的对数（即预计故障数）视为多个独立变量的线性组合。换言之，NBR假定预计故障数是这些独立变量的积性函数。

下列这些标量组成了我们所说的预测用*标准模型*。这些变量将作为每个文件的预测模型的输入数据，然后预测模型将输出该文件在版本N中的预测故障数量。

- 文件大小：文件中的代码行数（LOC）。
- 文件是前一个版本就存在还是新加入这个版本（二选一）。
- 在版本N－1中该文件的修改次数。
- 在版本N－2中该文件的修改次数。
- 该文件在版本N－1中所检测出的故障数量，无论是在测试还是发行后检测出。
- 文件所使用的编程语言。

如需标准的NBR模型的详细技术说明，请参见Weyuker等人的报告[9]。

在我们的实验中，我们用预测故障最频发的20%的文件在当前版本中所包含的实际故障数量除以该版本的实际故障总数，这样就能看出模型的预测结果有多成功。也就是说，如果我们预测的故障最频发那20%的文件包含了80个故障，而版本N的故障总数为100，那我们就可以说我们的模型准确地在20%的文件中识别出80%的错误。当然，当我们在正在进行的开发环境中应用这些模型的时候，开发人员并不知道版本N中具体有哪些文件会被预测为故障频发。

9.4　预测模型的复验和变体

我们必须在不同的项目中重复这些实验，才能证明这个方法有效。对于实验对象系统的相同

版本采取相同的统计模型一般来说无法给出什么新的信息，因为我们采用的预测算法是确定性的。因此，此处所说的重复实验指的是要么对同一系统的不同版本应用同样的模型，要么对多个不同的系统使用同样的模型。

两种重复实验我们都已做过。在表9-1中，我们展示了在研究中研究过的各系统版本数量。在这一节中，我们将讨论另一种形式的重复实验，即使用模型变体。

虽然上一节中介绍的NBR模型已经达到了很高的准确率，但我们还是继续研究了可能的改进方法。

在研究中，我们尝试在NBR中代入不同的变量，还用NBR和其他统计模型做了对比。这些实验表明，想要在标准NBR模型的基础上再进行改善非常困难。我们使用额外的变量来增加准确率的收获最多不过约1%，而没有任何的其他统计学模型能得出比NBR模型更准确的结果。

这样，除了对改进预测的方法进行了探索以外，这些实验还可以作为一个旁证，证明我们的故障预测方法确实是非常健全的。

9.4.1　开发人员的角色

很多软件开发人员相信，对于软件质量和错误预测来说，很重要的一点是谁或者有多少人最近修改过代码。他们通常认为，修改某段代码的程序员越多，这段代码就越容易出bug。当我们展示我们的预测方法以及标准NBR模型的时候，最常被问到的一个问题就是在模型中代入编写和修改代码的程序员的信息会不会让预测的结果更加准确。为此，我们决定对标准模型进行扩充，加入开发人员的信息，并将结果和不加入这些信息的标准模型得出的结果进行对比。

MR的数据库中包含了每个版本中的每个文件的每次文件修改所对应的开发人员ID。这让我们可以找出从头到尾有哪些开发人员修改过这个文件。可惜的是，数据库中并没有首次创建这个文件的开发人员信息。尽管如此，这些ID还是让我们可以在预测模型中加入下面三个变量。

- 累积开发人员数

  从版本1到版本N－1有多少个开发人员修改过这个文件 。
- 最近开发人员数

  版本N－1中有多少个开发人员修改过这个文件。
- 新开发人员数

  在版本N－1中有多少个开发人员是首次修改这个文件。

虽然在单独加入每个开发人员相关的变量时，预测的结果都只会得到少许改善，但是累积开发人员数的改善总是相对最大。所以，我们只展示了这个变量的结果，并将这些结果和原版的标准NBR模型的结果进行对比。

图9-1、图9-2和图9-3分别展示了维护系统A、B、C的信息。这些数字最初刊载在Weyuker 等人[9]的报告中。每个图都分成上下两部分，并按版本号对齐。图的上半部分是一个条形图，展示的是在每个版本中发现的故障总数。可以看出，bug总是集中在一小撮文件中。

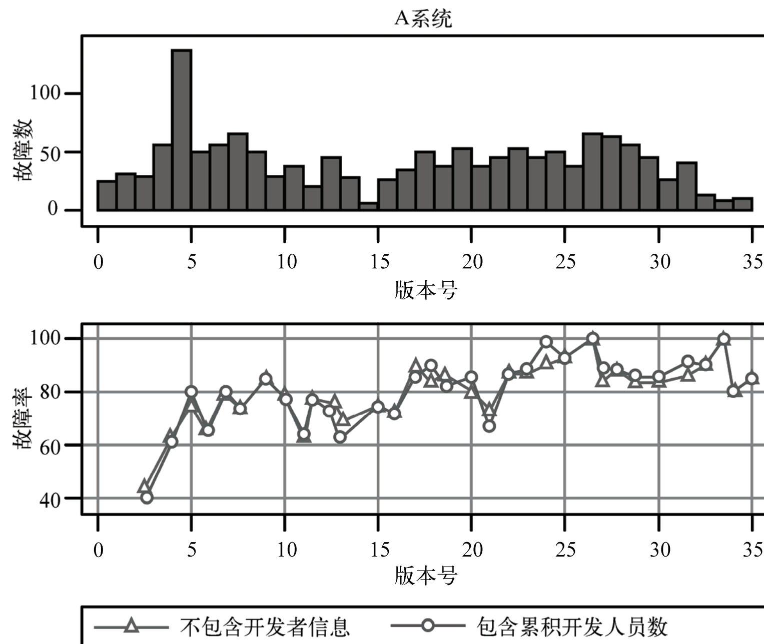


图9-1　加入与不加入累积开发人员数的预测结果对比：系统A中按版本号分类的前20%文件中的故障数和故障率

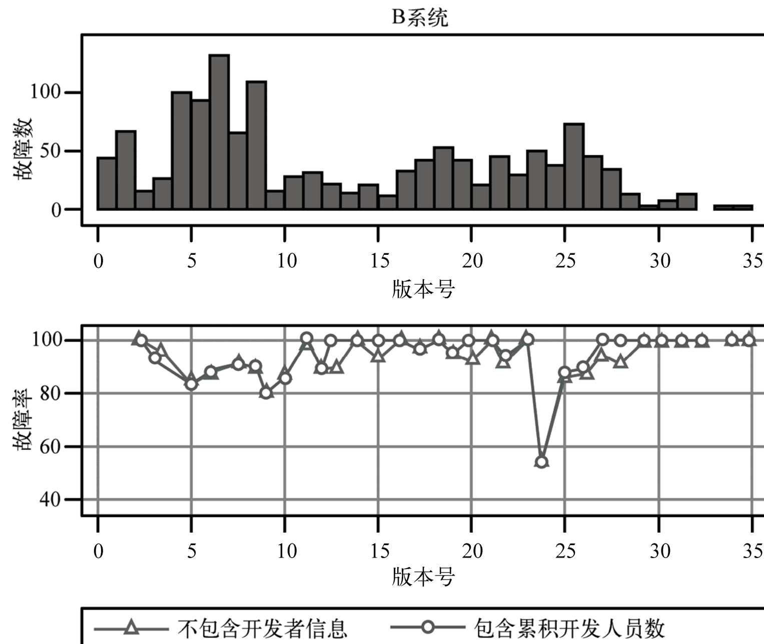


图9-2　加入与不加入累积开发人员数的预测结果对比：系统B中按版本号分类的前20%文件中的故障数和故障率

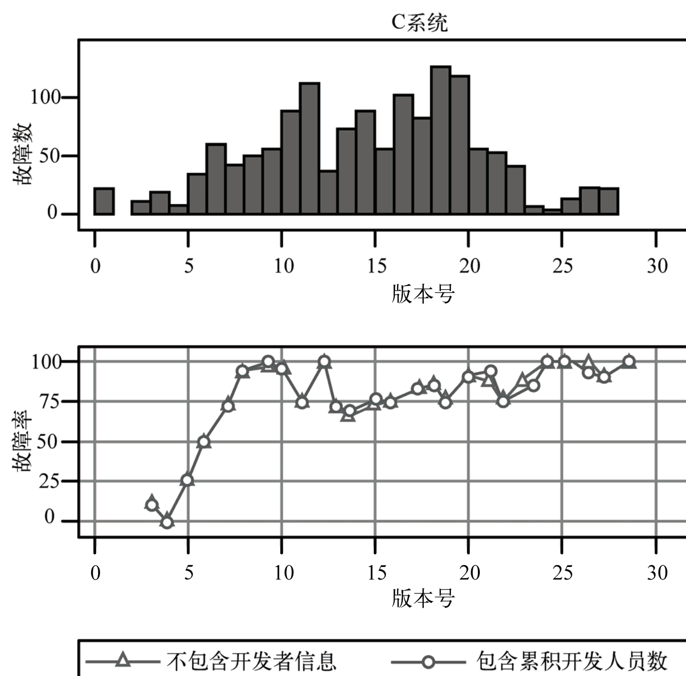


图9-3 加入与不加入累积开发人员数的预测结果对比：系统C中按版本号分类的前20%文件中的故障数和故障率

图的下半部分包括了两根线，用于对比标准模型和用累积开发人员数扩充过的模型的成绩对比。两根线都表示在预测出的故障频发的那20%的文件中实际包含的故障占总故障的百分比。

我们可以看到加或者不加开发人员信息实际上并没有多大差别。拿系统A来说，预测改善的平均值只有0.2%，而对于系统C来说则完全没有改善。甚至对于改善最大的系统B来说，改善程度也只有寥寥的1%。此外，我们还发现了一个有趣的现象，那就是这三个系统中总有那么一个版本，添加了开发人员信息反而让预测的准确度降低了。

若需要其余更详细的研究结果，请参见Weyuker等人的报告[9]。

9.4.2 用其他类型的模型来预测故障

虽然我们已经确定NBR模型可以识别出包含了绝大部分故障的一小撮文件，但是我们仍然想要将这个模型和其他的模型的结果进行对比。我们使用了3中不同建模方法：递归分组、随机森林以及贝叶斯加性回归树（Bayesian Additive Regression Trees，下称BART），来为表9-1中的3个系统创建了模型。随机森林和BART是递归分组的扩展，用于解决递归分组的局限性。

递归分组基于养成数据来构造二元决策树，如此，每个叶节点都代表了一组相似的输入值，

这些输入值所对应的输出值（即故障预测数）相互比较接近。我们可以这样来理解这个树，首先我们的目的是为新版本中的每个文件预测故障的数量。预测流程的输入是文件，而输出则是预测的故障数量。

每个文件都带有一系列的属性，包括其长度、故障及修改历史、编程语言等。这些属性都是预测流程的独立变量，或称预测指标变量。决策树将用这些属性值来为每个文件分别定义一条从根节点到子节点的路径。树的每个内部节点都将基于每个预测指标变量的值把输入分成两组。该输入值将按照该节点所对应的指标变量选择左分支或者右分支进行下一步。当输入值达到叶节点的时候，其预测故障数则是根据该节点上的所有养成数据所得出的平均故障数。

树的构造是从上到下的，首先是将所有的养成数据分为两组，分组将基于单独的一个指标变量，这个变量必须保证每组内的数据的均方误差降到最低。每个文件的养成数据的误差就是该文件的故障数和该节点的平均故障数的差距。每次分组产生的节点又会再被分成两组，分组同样基于均方误差最低的单个变量。这样的*递归分组*将不断重复，直到满足某个停止条件。

由于递归分组和回归分析的理念完全不同，所以这样的方法产生的模型和负二项式产生的模型有很大不同。

递归分组有个缺点，那就是决策树的构造非常依赖最早的几次分组。一旦养成数据被分配到了树的某一边，这些数据就无法再关联到树的另一边了，而另一边可能正好有非常接近的故障数。这个问题由随机森林来解决，方法是创建一大堆决策树，并用这些树的平均预测来作为输出。我们的每个故障预测森林中都包含500棵决策树。很明显，随机森林所需要的时间要比NBR或者递归分组的时间更多一些。

我们使用的最后一种方法称之为贝叶斯加性回归树（BART）。这个方法解决递归分组问题的方法是将预测分成小块，并分别对它们进行递归树的流程，最后再把结果汇总。我们在使用这种方法的时候，在每个预测模型中都使用了100棵递归树。

若需了解这些方法的详细信息以及我们在使用这些方法的过程中所发现的一些局限性问题，请参见Weyuker等人的报告[10]。我们使用了R[R Project Statistical Computing]程序库中对这种三种方法的实现来构建这些模型并进行预测：rpart包[rpart Package]对应递归分组模型，randomForest包[randomForest Package]使用的是随机森林来预测，而BayesTrees包[BayesTree Package]则是用BART模型来预测。

我们这次模型比较实证研究的对象分别是维护支持系统A、B和C，结果如表9-1、表9-2及图9-4所示。这些结果取自于Weyuker等人的原论文[10]，展示的是这几种预测模型找出的20%故障最多发的文件中所包含的故障百分比。条状图表示的是我们所预测过的系统的各版本平均值。图表显示，对于这三个系统来说，NBR和随机森林模型的效果比BART和递归分组模型要好很多。虽然NBR和随机数模型的预测结果并没有统计学意义上的差别，但是前者所需要的时间却比后者要少很多。所以，我们认为至少对于这些系统来说，NBR是最好的选择。

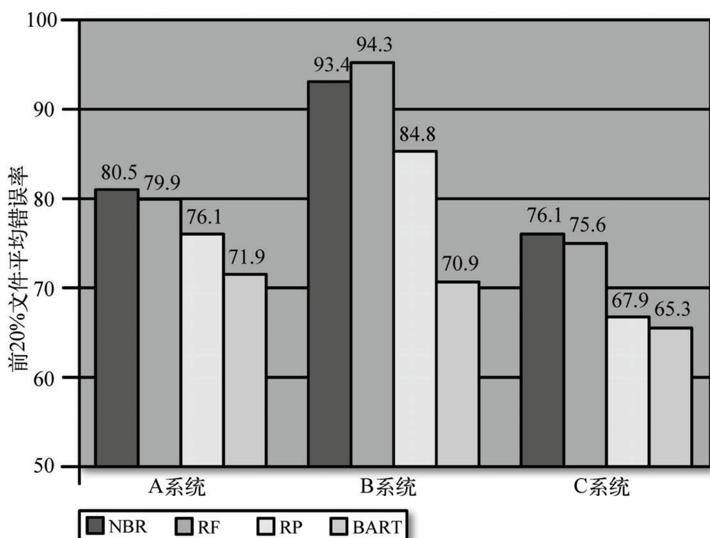


图9-4　四种不同模型的预测结果，针对系统A、B、C

9.5　工具的设计

由于我们的标准NBR的预测能力已经被证实，而参与者又都想要在实际项目中使用我们这项技术，所以我们决定设计一套自动化编程环境工具，让用户可以为即将进入系统测试阶段的文件生成下一版的故障预测结果。

我们做了一个可使用的原型工具，这个工具只需很少的专业知识或者用户输入。用户需要指出需要进行预测的系统中正在开发的版本，以及需要做预测的文件类型。此外，用户需要告诉这个工具如何来识别保存故障修复历史数据的MR数据库中的条目。故障的识别信息要么就包括MR所处的开发阶段，比如系统测试阶段，或者就包括MR的发起人，比如系统测试人员。

这个工具返回结果的形式是一系列的文件，这些文件将按照预测故障数降序排列。用户可以指定想要查看故障最多发的文件的百分比，也可以选择只查看某一类型的文件。比如说，用户可以只查看问题最多的10%的Java文件，或者查看问题最多的20%的C、C++、Java或SQL文件。预测结果的生成一般来说非常地快。

用户无需任何数据挖掘算法或者统计学的知识，也不必对这个工具的工作原理进行了解，直接就可以使用这个工具。

我们的不少开发和测试经理都已经看过了这个工具原型的实际运作过程，他们都希望能使用这个技术来为他们的软件系统做预测，以便能指导需求和设计的审查和帮助测试工作。

9.6　一些忠告

在设计我们的故障预测技术和工具的过程中，我们做了大量的实证研究。由于在商业环境中

很少见到针对多个系统反复进行的跨越数年的实证研究，所以我们认为读者也许会想了解在这个过程中哪些部分是最困难的以及造成困难的原因，这样他们就能理解为什么这样的证据虽然很难搜集，但是却至关重要。

- 到哪里去找系统来研究

  很多读者可能会觉得，你们不是为一个大公司工作吗，你们的公司不是有数以亿计甚至十亿记的代码同时运行吗，找个系统来研究还不容易？但事实并非想象的那么简单，尤其是在研究的初期。这有点类似于先有鸡还是先有蛋的问题。系统的管理者们不愿意让人研究他们的系统，并且在他们看来，这样做的理由有很多。

  首先他们需要担心的是你会占用他们的时间，而他们的项目又常常是迫在眉睫。如果他们花时间来回答你的问题，对他们来说并没有直接的好处，但是却有可能使他们无法按时部署软件系统。所以他们不愿意参与到研究中来，因为他们觉得这个事情是个风险很高的研究项目。很多研究项目确实是高风险的，因为这些项目停留在了理论阶段，参与者无法利用研究的结果。

  管理者们还担心研究人员会修改系统或者数据，然后对项目造成不良的影响。虽然我们可以保证只看不碰，但是他们还是会担心你会有意或无意地影响他们的系统。

  和其他领域的研究类似，我们的研究还有一个问题。比如，在医疗或者药品实验中，作为研究对象的人们很少能从研究中得到好处，不但如此，他们还必须被告知参与所可能带来的风险。

  实际上，我们的最早的参与者们实际上并没有获得任何好处，因为我们只是在寻找和故障高发文件联系最紧密的各类特征，而预测模型的设计也刚刚开始。

  即使在我们有了一个看上去成功的预测模型之后，也还是需要几个系统来做验证。在验证完成之后，我们才开始创建一套自动化工具。在这个工具开发完成之前，要做预测就必须了解数据挖掘和统计学的相关知识，还必须对统计工具（如R）有一定了解。这已经超出了普通软件开发人员以及测试人员的能力范围。

  一旦你用这个技术预测成功过几次了之后，想要找项目来参与研究就不难了，当然，首先还是要有一些有奉献精神并且甘当小白鼠的项目。这通常需要你和项目人员在文前的工作中建立的良好关系，以此来说服他们你了解项目在时间上的限制，还有就是你了解在研究过程之中绝不能影响项目。

- 预备研究非常困难而且耗费时间

  正如前面所说的，在我们正式研究故障预测之前，我们还必须确认故障不是均匀分布的，还得确定哪些特征和故障文件的关系最紧密，这些都需要做很多预备研究。当然，我们可以用很多坊间经验来作证，尤其是证明和帕累托分布相关的假说，而且早前已经有一些研究考虑了和高故障率相关的因素，但是个人经验并不能和对大型商业系统进行的专业研究相提并论。我们必须在研究的环境中确认我们能观察到这一现象。最后，对不同特征的重要性所做的早期研究中，有些得出了矛盾的结果，而有的根本没有把代码相关特征以及历史相关特征都考虑进来。

我们大概花了两年时间来做这些预备研究，然后我们才觉得准备好为第一个系统构建统计预测模型。

- 取得并分析数据非常困难且耗费时间

  当我们确定了最关键的代码及历史特征之后，我们还必须了解软件系统所使用的变更管理和版本控制系统，并确定如何来提取必要的数据。在研究第二个系统的过程中，我们发现即便是同一个数据字段，不同的系统也有着不同的使用方式，或者有时候使用得并不到位。这有助于我们确定哪些数据有用，哪些没用。随后，我们需要编写一些脚本来执行数据提取的工作，并建立原始预测模型。

  如此，又花了一年时间之后，我们才终于能够用定制的预测模型对第一个系统进行初步预测。

- 怎样才算成功很难定义

  我们的模型将为每个文件关联下一版本的预测故障数。我们考虑了多个方法来评断这些预测结果的好坏。

  一种方法是直接对比每个文件的预测故障数和实际故障数。就我们研究的第一个系统来说，两个数字并不紧密匹配。然而，文件故障数的相对顺序却常常是非常相似的。在和项目人员们讨论之后，我们发现最好的评估办法是如本章早前所说的前20%的文件。

  我们考虑了多种在其他研究文献提出的指标，但是最后发现这些指标并不适用于我们的研究，因为它们是设计用于预测二元决策（比如一个文件是否包含故障），而不是像我们的模型那样为文件进行排序。我们继续评估了多种可选的评估标准，以确保我们使用的方法最适合于我们的需求。

- 要获得“ 客户 ”是很难的

  我们完成了数个大型的实证研究并一致获得良好结果，还开发了一整套自动化的工具，尽管如此，我们还是很难说服项目领导去修改开发方法并在开发流程中融入我们的预测模型和工具。我们最近找到了一个项目，这个项目的管理层认为这项技术非常有用，我们正在讨论把技术转让给他们。使用这些工具会否影响用户识别故障的类型以及数量，还有待观察。

  那么我们的底线是什么？在这项研究上，我们已经花了8年时间。有必要花那么多时间吗？我们希望本章中谈到的内容会让你相信花这些时间是非常值得的。想要在我们这样大的范围内收集证据通常需要很多时间，尤其是在技术正在开发和成熟阶段的时候，因为在这些阶段的大部分事情都必须人手动完成。不过，现在已经有了大量的证据证明我们的预测模型足以应对各种不同的环境和条件，而且又有了一套完全自动化的工具，我们相信，下面该是它光芒四射的时候了！

9.7 参考文献

[1] [BayesTree Package] The BayesTree Package. *http://cran.rproject.org/web/packages/BayesTree*

[2] [McCullagh and Nelder 1989] McCullagh, P., and J.A. Nelder. 1989. *Generalized Linear Models*, Second

Edition. London: Chapman and Hall.

[3] [Ostrand and Weyuker 2002] Ostrand, T.J., and E.J. Weyuker. 2002. The Distribution of Faults in a Large Industrial Software System. *Proc. ACM/International Symposium on Software Testing and Analysis* (*ISSTA2002*): 55-64.

[4] [Ostrand et al. 2005a] Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2005. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering* 31(4): 340-355.

[5] [Ostrand et al. 2005b] Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2005. A Different View of Fault Prediction. *Proc. 29th IEEE Annual International Computer Software and Applications Conference* (*COMPSAC 2005*) 2: 3-4.

[6] [R Project Statistical Computing] The R Project for Statistical Computing. *http://www.r-project. org/*

[7] [randomForest Package] The randomForest Package. *http://cran.rproject.org/web/packages/ randomForest*

[8] [rpart Package] The rpart Package. *http://cran.rproject.org/web/packages/rpart*

[9] [Weyuker et al. 2008] Weyuker, E.J., T.J. Ostrand, and R.M. Bell. 2008. Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models. *Empirical Software Engineering Journal* 13(5): 539-559.

[10] [Weyuker et al. 2010] Weyuker, E.J., T.J. Ostrand, and R.M. Bell. 2010. Comparing the Effectiveness of Several Modeling Methods for Fault Prediction. *Empirical Software Engineering Journal* 15(3): 277-295.

第10章

架构设计的程度和时机

Barry Boehm

本章会通过介绍一些研究来帮助项目和开发组织确定什么时候该使用敏捷的开发方法，什么时候该使用基于架构设计的开发方法。虽然软件及软件密切相关的项目的管理者们都知道架构设计很重要，但是他们手头的资源都很有限，所以他们常常会问："到底架构设计要做到什么程度才够呢？"（参见10.2节。）

本章总结概括了两类证据（这些证据是由我和其他的研究人员们在40年软件实践的经验中累计起来的），以便能为有上述疑惑的管理者们提供一些实践上的指导方针。此外，我们还将讨论一下我们对这些问题的一些看法。我们的研究想要回答的问题可以概括为如下内容。

- 随着项目的时间的推移和规模的变大，你修改或者修正缺陷的成本会相应地增加多少？
- 在开始正式的产品开发之前，你应该投入多少在早期架构设计以及实证基础上的项目回顾上？

在本文中，"架构设计"并不是指为软件制作详尽的计划大纲一类的事情，而是指所有对于创造和维护软件产品来说至关重要的各种事务，包括现场调查、运营分析、需求与机遇分析、成本分析、需求及架构的定义、规划和调度、可行性的确认和验证等——即《系统架构设计》（*System Architecting*）一书中的对于架构设计的定义[26]。

10.1　修正缺陷的成本是否会随着项目的进行而增加

对于我们第一个问题的重要性，敏捷开发的领军人物Kent Beck在他的著作《解析极限编程》①（*Extreme Programming Explained*）[1]中已经有过说明。

图10-1展示的是"XP（极限编程）的技术前提"。如果对软件进行修改的成本"随着时间的推移而缓慢地增长……那么你就会把各种决策尽量推后……并只实现你需要实现的"。

但是，"如果扁平的成本曲线意味着我们可以使用XP，那么陡峭的成本曲线则意味着我们无法应用XP。"

① 该书由电子工业出版社2006年出版。——编者注

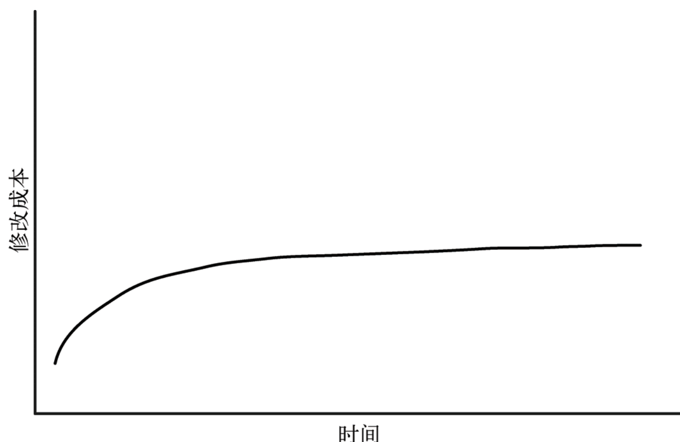


图10-1　修改的成本可能不会随着时间的推移而急剧上升

所以，我们可以问：XP或者其他的敏捷开发方法能否帮助我们保持修改成本曲线的扁平？我们还可以问：还有没有什么别的办法可以让成本曲线扁平化，以便项目后期的修正不会威胁到项目的总预算和交付时间？还有就是，对于那些有着陡峭成本曲线的项目来说，如何减少后期的修改？这一章将回答这些问题。

10.2　架构设计应该做到什么程度

如果说陡峭的成本曲线使我们无法应用敏捷的开发方法（如XP），那么在项目的准备阶段我们应该做多大投入？有的决策者这样打比方："修房子的时候，我们付给建筑师的钱差不多等于房子价值的10%，我们也会按照同样的比例付给软件架构师。"但是到底百分之多少对于软件来说才是合适的？10%有可能超出太多，也有可能远远不够。很多"铁公鸡"决策者认为软件架构设计并不直接生成软件产品，所以会尽量减少这部分的成本，但是这往往会导致后期返工的增加，并且反而让项目的时间和成本超出预算。这样看来，任何与"架构设计应该做到什么程度"相关的证据对于企业或者项目决策者来说都会非常有价值。

成本-修正增长比

在20世纪70年代，很多组织，如IBM[18]、GTE[16]，TRW[2]和贝尔实验室的Safeguard项目组[31]，都对大型软件项目中不同阶段的修改（或缺陷修正）的成本做了研究。这些研究较为一致地表明，交付后的修改成本差不多是需求确定阶段的修改成本的100倍。图10-2是从我1981年的书《软件工程经济学》（*Software Engineering Economics*）[4]中提取出来的，总结了这些研究以及其他一些相关研究的结果。这表明，虽然对于大型系统来说确实是有100：1的比率，但是对于小一些的系统（2000～5000行源代码）来说，5：1的比率更有代表性[3]。

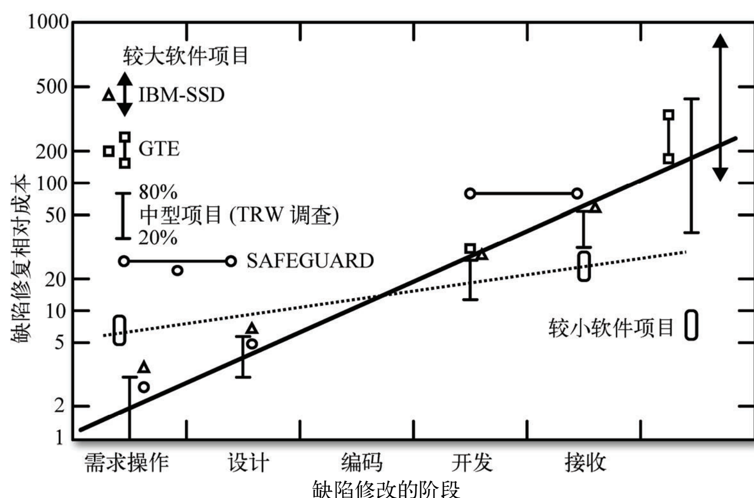


图10-2　不同阶段的缺陷修复的相对成本（20世纪70年代）

不过，我们还不太确定对于这些影响成本的因素到今天还会不会有同样的影响。为了探讨这个问题以及一些相关的其他问题，由美国国家科学基金会出资，并由美国马里兰大学和南加州大学设立的实证软件工程中心中的有相关经验的人员对文献进行了检索，并组织了三个在线学术讨论会[30]。这些研究结果基本确认了大型项目的100∶1的比率，实际的比率在117∶1和137∶1之间，而另一份1996年进行的研究显示，比率的范围在70∶1～125∶1之间[23]。不过，相关的学术讨论会的讨论结果表明，在大型项目的早期需求阶段进行的投入，以及对设计的确认和验证可以把成本–修改的增长曲线从100∶1降低至20∶1。有一个成功的案例做到的甚至更好——我们在下一节将会介绍一个百万行的大型CCPDS-R项目，据我所知这个项目是唯一一个符合图10-1中那种平稳曲线的项目[29]。

虽然Beck和其他人都使用了一些坊间数据来证明敏捷方法可以让项目更符合图10-1那样的曲线，但是对于小型敏捷项目进行的实证研究却没能证明这一论断。最近一份针对两个使用了部分XP实践（比如结对编程、测试先行、现场客户）并且未违反XP的其他实践的Java商业项目的实证研究提供了一些修改成本相关的实证证据，如图10-3和图10-4所示[21]。

虽然在这些研究中没有具体的修改数量以及每个修改的工作量的信息，但是我们从项目1中的事例（story）总工作量比率中还是能看出，重构的工作量的每事例平均增长在6%左右（从第一个事例占0%的工作量到最后第六个事例完成占差不多35%的工作量），而解决错误的工作量的增长则在4%左右（从0%到最后大概占25%）。项目2的相应数据大约是5%和3%。这样的增速比较平稳，虽然明显不如敏捷的行家们的坊间经验那么夸张，但是却更可能代表了使用XP的主流经验。从项目1到项目2，增速有所放缓，这意味着人们对于XP有着可觉察但是并不特别突出的学习曲线效应。

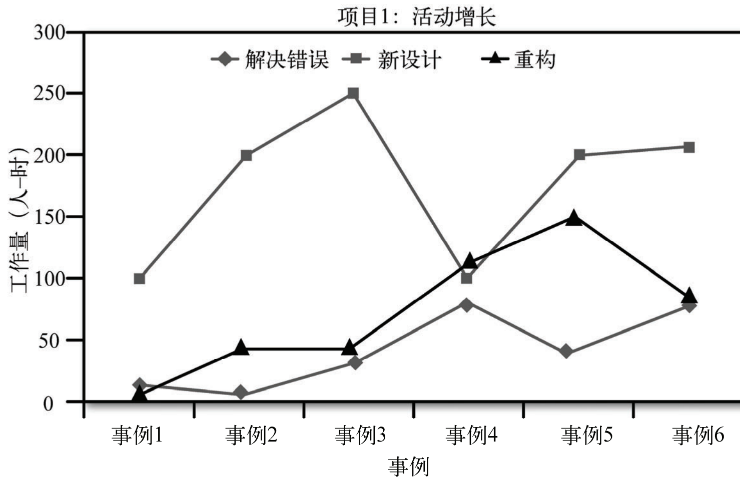


图10-3　项目1：修改成本增长图

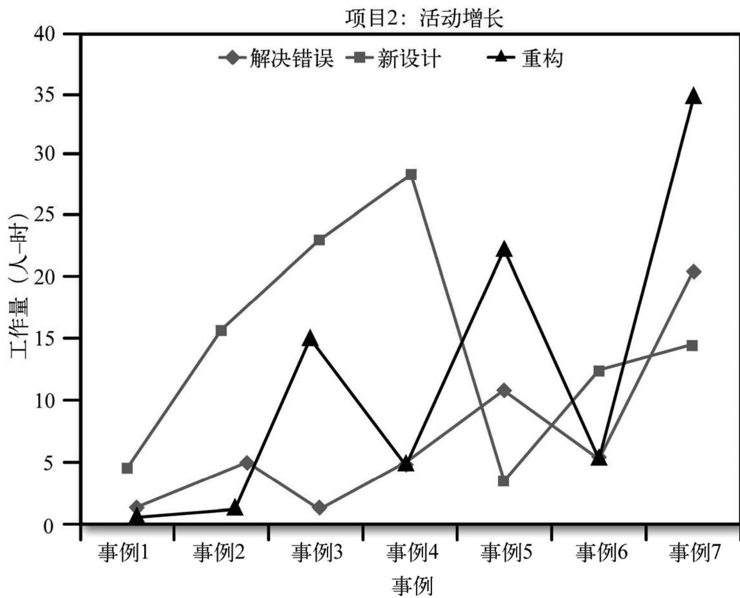


图10-4　项目2：修改成本增长图

Reifer[28]也在研究中引用了一些具有类似特点的私有实证数据。此外，我们还从Elssamadisy和Schalliol的研究[31]中关于ThoughtWorks公司的XP租约管理器项目的经验中找到了一些佐证。当项目的规模超过50个人、1000个事例以及50万行代码的时候，他们发现“虽然每个人都声称这些事例卡片上的任务是按时完成的（即30天内），但是实际上却还是经过了整整12个星期的开发才终于可以交付高质量的应用程序给客户”。

10.3 架构设计的成本–修复数据给予我们的启示

我们需要在项目最初就把架构设计独立出来作为一系列并行系统以及项目定义，以便确定它能不能减少后期的返工及各种相关项目成本。在这一节中，我们将展示如何把架构设计和成本联系在一起，并在本章后面的部分继续用图表来解释何时做架构设计，以及在不同类型的项目中应该把架构设计做到什么程度。

10.3.1 关于COCOMO II架构设计和风险解决系数的基础知识

这一节的内容基于我和同事们的一份长期研究。我们花了数十年的时间设计了一个用于估算软件项目的成本及开发时间的模型，我们称之为建设性成本模型（Constructive Cost Model，COCOMO）。在把对风险的评估加入了这个模型之后，我们成功地展示了架构设计的价值以及在本章后面部分将会详述的“做到什么程度才够”的问题。

1. 规模经济与规模不经济

我们原始的COCOMO[4]模型中并没有加入衡量架构设计全面度的指标，因为当时我们没有考虑到团队是否能处理好规模不经济的问题，而大多数团队都会遇到这个问题。

那么，到底什么是规模不经济（还有规模经济）？这些都是经济学的术语，用于把产品的产量和单位成本联系起来。当生产更多的产品可以让产品的单位成本降低时，那么我们说这是规模经济。反之，当生产更多的产品导致单位成本升高时，我们就说这是规模不经济。

我们看重这些指标的原因之一，是因为软件产品现在越来越大，而我们希望能够尽量减少规模不经济。另外一个原因是因为软件开发人员和硬件开发人员对于这些术语有着截然不同的理解。硬件开发人员把产品看做像车或者手机一样的实体：生产得越多，单位成本自然就更低。换句话说，硬件行业一般属于规模经济。对于软件来说，大量制造副本的成本几乎是零，而真正和成本相关的产量单位则是源代码行数（SLOC）。就这个产量单位来说，新的SLOC越多，每单位的成本反而越高，也使得软件项目常常会遭遇规模不经济。

要深究其原因的话，我们可以考虑一下创建两个分别包含10000条新SLOC的软件组件的成本。如果开发成本是$20/SLOC，那么每个组件的成本就是10000 × $20=$200000。

但是如果我们要把这两个组件组合成一个单独的产品呢？除了每个组件所需的$200000之外，还会有各种额外的开销，比如让它们可以进行交互的设计成本，将它们整合成一体的成本，对整合后的系统进行测试的成本，以及解决整合后可能的各种缺陷的成本等。

虽然说交付的SLOC仍然是20000，但是成本却远不止2 × $200000=$400000，每SLOC的成本增加了，造成了规模不经济。软件开发人员们往往很难向硬件思维导向的管理者解释这个问题，因为在硬件思维管理者的心中产量越大，单位成本应该越低。

2. 通过架构设计和风险处置的方式来减少返工

在原始的COCOMO模型中，最接近架构设计全面度的指标是现代编程实践，包括了自上而下的开发方式、结构化编程以及设计和代码检查等。通常的看法是，无论对于大型项目还是小型项目来说，这些实践都有着相同的影响。但是，我们认为规模不经济却是直接取决于项目的开发

模式。例如下列模式。

- 一个使用有机模式（Organic-mode）的低优先级的项目的规模－工作量增长指数是1.05。这意味着如果产品的规模增加一倍，工作量将增长2.07倍（即$2^{1.05}$）。
- 一个使用嵌入模式（Embedded-mode）的至关重要的项目的增长指数是1.20，这意味着产品规模增加一倍的话，工作量将增长2.30倍。

我们把COCOMO数据库中的63个项目的所使用的模式的规模和工作量的增长数据对应起来得到了上述指数。

随后，我们于20世纪80年代在TRW公司的研究和经验显示，改善后的实践可以减少某些造成软件开发的规模不经济的问题（比如返工），而全面的架构设计则是改善的重要措施之一。比如，有些大型的TRW软件项目由于架构设计工作以及风险应对措施没有做够而导致了很高的返工成本[6]，较小的项目也出现了一定的返工成本。

对于项目的缺陷（主要的返工成本来源之一）修复成本数据的分析显示，20%的缺陷造成了80%的返工成本，而这20%的缺陷主要是由于缺乏足够的架构定义以及风险应对措施造成的。

如图10-5所示，在TRW项目A中，大部分的返工都是由网络操作系统使用了不完整的架构造成的。由于在设计架构的时候忽视了操作系统可能不支持“在网络处理器功能失常时将系统故障转移”的功能，而这正是这个项目所必须的。当在系统测试中发现了这个问题之后，它就会变成一个“架构破坏者”，导致已经开发好的软件又不得不进行一系列高成本的返工。在项目B中有一个类似的“架构破坏者”，那就是在项目晚期才发现的处理超长消息（即超过100万个字符）的重要性。B项目不完整的架构假设所有的消息都会很短并且在封包交换网络架构中很容易处理。（做出这个假设有部分原因是需求声明比较模糊，即“系统应该圆满地发送所有提交给它的信息”。）

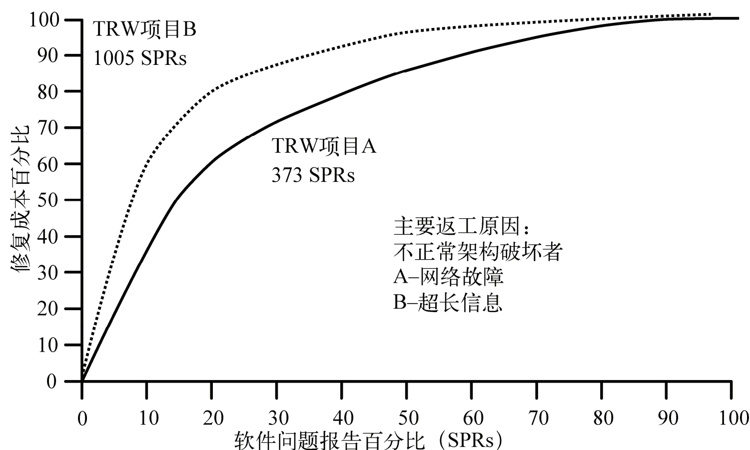


图10-5　对应高危元素的较为陡峭的修复成本增长曲线

这样的结果使得TRW制定了各种政策，要求开发人员在项目的初步设计评审（Preliminary Design Review，PDR）阶段对所有的需求做完整的风险分析。由于TRW采用了Ada编程语言，而这个语言又可以确保模式规格的一致性，这条风险分析的政策也延伸扩展成了Ada流程模型（Ada

Process Model），并要求在PDR之前，软件的架构就必须通过Ada编译器模块一致性检查[29]。这就让后来的软件项目可以在程序员写代码和做单元测试之前就做好大部分的系统集成工作。

3. 一个成功的例子：CCPDS-R

由于吸取了在上一节中所描述的经验教训并在PDR之前消除了架构上的风险，在后来的项目中，由架构破坏者造成的后期的返工大大减少了，而修复成本曲线也更平稳了。在这里有个很好的例子，那就是Royce报告[29]中所描述的“指挥中心处理及显示系统–替换版”（Command Center Processing and Display System-Replacement，CCPDS-R）项目，图10-6展示了其平稳的修复成本曲线。这个项目在规定时间内用规定的预算交付了超过100万行Ada代码。

在CCPDS-R项目中，人们修改了固定流程的瀑布开发模式，延缓了里程碑版本的发布日期，以便于早期就采用基于风险评估的螺旋型开发模式。举例来说，这个项目预计用35个月来完成第一版的交付，而其PDR是在项目的第14个月进行的的，花费了第一版预算的1/4，包括了设计和验证其高风险的软件组件，比如其网络操作系统以及用户界面的关键部分。

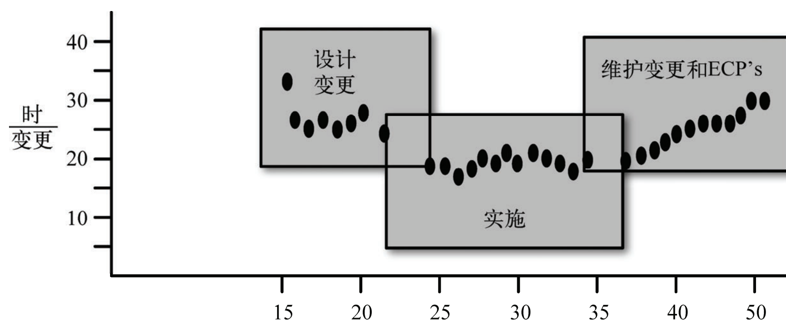


图10-6　降低修复成本成长率：CCPDS-R[29]

10.3.2　Ada COCOMO及COCOMO II中的架构设计以及风险应对系数

图10-6中所展示的大型项目中平稳的修复成本曲线证实了对于架构设计和风险处理更多的重视降低了大型项目中的返工和规模不经济的情况。从1987年到1989年，TRW为使用Ada流程模型的大型关键项目设计了一个专门版本的COCOMO，称之为Ada COCOMO[11]。它降低了Ada流程模式所具备的1.20的规模–工作量增长系数。而由于改善了架构设计以及风险应对，它还减少了项目的规模不经济的情况，降低了对成本的估算，并帮助政府和业内的管理者从相对僵化、流水线式、基于文档的旧式流程，转化到更多基于风险的并行开发流程。

1. Ada流程模式如何推动基于风险的并行开发软件流程

Ada流程模式以及CCPDS-R项目表明，我们完全可以改造顺序式的瀑布流程模式，使其成为一个更现代的基于风险的模式，并同时进行需求、架构以及规划的活动。这个模型的审查标准主要关注软件组件的兼容性以及可行性。

随后，研究人员们把这些实践整合成了通用的软件开发以及系统工程的流程，以用于各类需要密集软件开发的系统。这些模型强调的是基于风险的并行开发，以及相应的里程碑版本审核判

断条件[5]。这些模型包括如下内容。

- Rational统一流程（Rational Unified Process，RUP）[29] [19] [20]。
- USC基于模式的系统架构设计以及软件工程学（Model-Based System Architecting and Software Engineering，MBASE）模式[9] [10]。这个模型相应的又加入了：
  - 风险驱动的并行工程学共赢螺旋模型（The risk-driven concurrent engineering WinWin spiral model）[12]。
  - Rechtin并行工程系统架构设计方法[26] [27]。

RUP和MBASE都使用了参照点式里程碑版本集来为项目分阶段，这些里程碑版本集包括生命周期目标（Life Cycle Objectives，LCO）和生命周期架构（Life Cycle Architecture，LCA）。这些里程碑版本是由USC软件工程中心及其分支的30个政府及行业机构所参与的研讨会选定，并用来作为COCOMO II成本及日程安排估算的阶段分隔标准。LCO和LCA的里程碑式参照点的通过/不通过的条件是如下所示。

- 开发人员提供证据，并由独立的专家组验证，证明系统是按照既定的架构来实现的，其必须：
  - 满足所有的需求——功能、界面、服务等级以及升级发展；
  - 对可操作理念的支持；
  - 可以在计划的预算和日程内完成；
  - 能产生足够的投资回报率（Return On Investment，ROI）；
  - 让所有与项目成功密切相关的各方都能满意。
- 所有重大风险都有对应的风险管理方案来解决。

最近，MBASE方法已经扩展成了增量承诺模式（Incremental Commitment Model，ICM），可以全面地用于系统及软件的开发。这种模式除了使用里程碑式的参照点之外，还使用了可行性理论，用于在系统的架构设计、需求、运行原理、计划以及商业考量中同步和稳定硬件、软件和人三个因素[25] [7] [8]。有效的可行性理论应该包含经过架构设计的权衡以及可行性分析（如Clements[15]和Maranzano[22]讨论的那种）得出的关于可行性的证据。

可行性证据的不足表现为项目遭受损失的不确定性或者可能性，我们可以把这些概率乘以项目损失的规模，就可以得出风险曝光度（Risk Exposure）。主要的项目关系方可以使用风险曝光度来确定是否继续进入下一阶段（即风险可接受）、跳过下一阶段（即风险微不足道）、延长当前的阶段（风险较高但仍可控）或者终止/重新设计项目（风险不可接受）。要想确保我们能获取足够的风险信息，搜集可行性证据就必须作为高优先级的项目目标之一，而不是作为一个随时可能因为预算和时间的原因被丢弃的可做可不做的附属任务。

2. COCOMO II中的架构设计和风险对应（RESL）系数

为了说服项目经理们加大对可行性证据搜集的投入，我们就必须提供实证证据来证明花在可行性证据上的时间将会带来可靠的投资回报（ROI）。我们用了161个具有代表意义的项目来为COCOMO II成本估算模型做校准的时候，确定了值得搜集可行性证据的项目必须具备的条件。

在1995～1997年间，在USC及其30个政府和行业内的分支机构组成的研讨会中，COCOMO II

软件成本估算模型的定义[13]得到了发展。其规模不经济系数依赖于架构设计及风险对应（RESL）系数以及其他四个规模系数：能力成熟度模型（Capability Maturity Model）中的成熟度水平，开发人员–客户–用户团队凝聚力（Developer-customer-user team cohension），优先性（Prededentedness），以及开发灵活度（Development Flexibility）。

对于RESL级别表的定义见表10-1中的七个组成系数。正如在 “Ada流程模式如何推动基于风险的并行开发软件流程”那一节中所展示的那样，架构设计和风险应对包括了并行系统的运行概念、要求、计划、商业考量、可行性理由以及架构，也就是说包括了软件系统工程中的大部分关键元素。

表10-1 RESL级别表

特　征	非 常 低	低	中	高	非 常 高	极　高
风险管理计划指出了全部的重大风险并用PDR或者LCA的方式确立了解决这些风险的里程碑式参照点	无	很少	一些	普遍	大部分	完全
通过PDR或LCA的方式设置的日程安排、预算以及内部里程碑参照点符合风险管理的计划	无	很少	一些	普遍	大部分	完全
在已经有了大体目标的情况下，项目组分配了百分之多少的开发时间在架构设计上	5	10	17	25	33	40
项目组总体架构师人数和所需人数的百分比	20	40	60	80	100	120
用于解决风险、设计和验证架构的工具的支持度	无	很低	一些	好	很好	完整
关键架构设计元素（任务、UI、现成解决方案、硬件、技术和效率等）的不确定性	极高	很高	中等	少量	很少	非常少
风险的数量以及重要程度	>10重要	5～10重要	2～4重要	1重要	>5不重要	<5不重要

每个为COCOMO II的数据库提供数据的项目在计算其RESL级别的时候都使用表10-1来作为参考。COCOMO II的研究人员和项目人员在收集数据的过程中一同确定了表格中每一行数据的相对权重。COCOMO II数据库中的161个项目的RESL级别大致是按照正态分布，如图10-7所示。

我们将COCOMO II的数据库中的161个软件项目中的专家意见和对规模、工作量以及成本评级的多元回归分析用贝叶斯定理（Bayesian）的方式综合起来，用于确定项目的RESL对于其规模不经济系数的影响。这些项目涵盖了商业的IT应用程序、电子服务、电讯、中间件、工程及科学、指挥控制、实时流程控制等多种领域。项目的规模从2.6到1300千行不等，每千行等同于源代码行（KSLOC），其中13个项目低于10 KSLOC，5个项目超过1000 KSLOC。用等同的源代码行数作为标准可以衡量软件的重用度，以及需求的变动度。

我们将专家们对于COCOMO II中带动成本的参数的平均值及标准差的意见作为贝叶斯定理的先验值，相对的，把通过对历史数据多元回归分析后得出的平均值及标准差作为贝叶斯定理的

后验值。使用贝叶斯定理的方法综合各方的数值之后，我们将得到专家意见及历史数据的加权平均值，对于那些标准差较小的参数值，权重就会更高。详细的方法及公式请参见关于COCOMO II的专著第4章[13]。

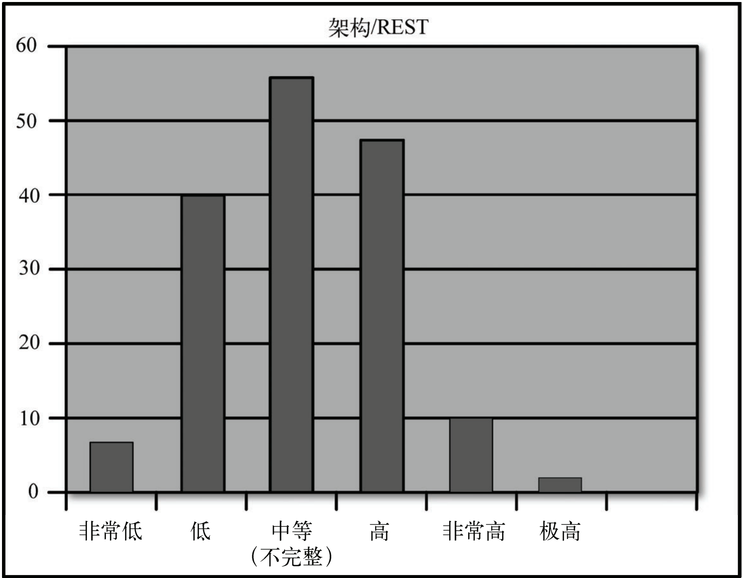


图10-7　COCOMO II数据库中的161个项目的RESL评分

3. 采用了架构设计和风险应对之后的改善

我们对于RESL级别的计算验证了我们的假设，即在软件开发中，如果架构设计以及风险应对没有做足（即系统工程没有做足），就会导致项目工作量增加，因为在开发后期项目组将不得不投入大量的精力来返工重做，以克服架构设计上的缺陷并解决研发后期出现的风险。我们还假设，依照我们对软件项目规模不经济的认识，对于越大的项目，返工的比率会越大。

对这些项目的RESL级别以及另外22个COCOMO II模型中影响成本的参数进行的回归分析产生了统计上显著的结果，证明了我们的假说。我们对这161个项目进行研究的结果表明，“非常低”与“极高”RESL级别的项目之间的成本−规模增长指数差距有0.0707。这对于10 KSLOC的项目来说相当于多了18%的额外工作量，而对于超大型的10000 KSLOC的项目来说，就是92%的额外工作量。

图10-8总结了这些分析的结果。研究表明，至少在这161个软件项目中，规模越大，增长的成本和工作量就相对地越大，这与COCOMO II模型中另外22个影响成本的参数无关。我们通过对另外22个参数进行的回归分析证明了这一点。RESL参数的统计显著性是2.084，超过了23个变量以及161个数据点的统计显著性值1.96，如表10-2所示。此外，配对相关度分析显示，RESL及其他所有变量的相关系数没有超过0.4。

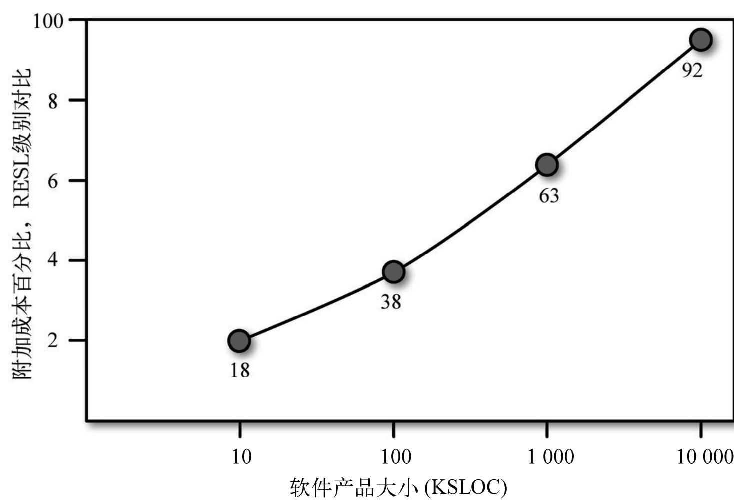


图10-8 减免软件系统设计所带来的附加成本

表10-2 COCOMO II回归分析结果

数据集=COCOMO II.2000
反应=log[PM]
系数估计

标 记	估 计	标 准 差	t-值
Constant_A	0.961552	0.103346	9.304
log[SIZE]	0.921827	0.0460578	20.015
PMAT*log[SIZE]	0.684836	0.481078	1.424
PREC* log[SIZE]	1.10203	0.373961	2.947
TEAM*log[SIZE]	0.323318	0.497475	0.650
FLEX* log[SIZE]	0.354658	0.686944	0.516
RESL* log[SIZE]	1.32890	0.637678	2.084
log[PCAP]	1.20332	0.307956	3.907
log[RELY]	0.641228	0.246435	2.602
log[CPLX]	1.03515	0.232735	4.448
log[TIME]	1.58101	0.385646	4.100
log[STOR]	0.784218	0.352459	2.225
log[ACAP]	0.926205	0.272413	3.400
log[PLEX]	0.755345	0.356509	2.119
log[LTEX]	0.171569	0.416269	0.412
log[DATA]	0.783232	0.218376	3.587
log[RUSE]	−0.339964	0.286225	−1.188
log[DOCU]	2.05772	0.622163	3.31
log[PVOL]	0.867162	0.227311	3.815
log[APEX]	0.137859	0.330482	0.417
log[PCON]	0.488392	0.322021	1.517
log[TOOL]	0.551063	0.221514	2.488
log[SITE]	0.674702	0.498431	1.354

10.3.3 用于改善系统设计的投入的ROI

在前一节中，我们证明了软件架构设计的重点应该是同时对系统需求、架构、规划、预算以及日程安排等使用风险驱动的方法进行计划。此外，还需要在原型设计、建模以及分析上加大投入，以保证项目的前后一致性和可行性。风险承担者在项目的每个阶段都进行审查，并建立对下一个阶段的开发的信心，对于项目的成功至关重要，我们在本章前面部分已经讨论过了。

图10-8中展示的用COCOMO II的方式来计算RESL级别的结果让我们可以确定这种投入的ROI，即将架构设计以及风险应对所需要的额外的工作量和不同规模的软件项目能够凭此而节约的成本进行对比。表10-3总结了从1万到1000万行规模的软件系统的结果。表格之后是我们计算这些值的具体方法。

表10-3 软件系统设计/RESL的ROI

COCOMO II RESL等级	非常低	低	中	高	非常高	极高
RESL时间投入%	5	10	17	25	33	>40（50）
工作量水平	0.3	0.4	0.5	0.6	0.7	0.75
RESL成本投入%	1.5	4	8.5	15	23	37.5
投入增量		2.5	4.5	6.5	8	14.5
返工工作量的规模增长指数	1.0707	1.0565	1.0424	1.0283	1.0141	1.0
10 KSLOC的项目						
附加的工作量%	17.7	13.9	10.3	6.7	3.3	0
效益，成本增量		3.8，2.5	3.6，4.5	3.6，6.5	3.4，8	3.3，14.5
ROI增量		0.52	−0.20	−0.45	−0.58	−0.77
100 KSLOC的项目						
附加的工作量%	38.4	29.7	21.6	13.9	6.7	0
效益，成本增量		8.7，2.5	8.1，4.5	7.7，6.5	7.2，8	6.7，14.5
ROI增量		2.48	0.80	0.18	−0.10	−0.54
1000 KSLOC的项目						
附加的工作量%	63	47.7	34.0	21.6	10.2	0
效益，成本增量		15.3，2.5	13.7，4.5	12.4，6.5	11.4，8	10.2，14.5
ROI增量		5.12	2.04	0.91	0.42	−0.30
10000 KSLOC的项目						
附加的工作量%	91.8	68.3	47.8	29.8	13.9	0
效益，成本增量		23.5，2.5	20.5，4.5	18.0，6.5	15.9，8	13.9，14.5

表10-3的前几行展示的是所有项目的综合统计，表格内容解释如下。

- RESL时间投入%

  这是最重要的指标，所有的值都相对这个指标来进行衡量。这个值代表了对于每个RESL评价等级来说花费在架构设计上的时间的百分比。

- 工作量水平

  在该RESL等级下的项目成员平均工作量占总工作量的比重。这个值的结果看上去和软件项目早期可以观察到的瑞利分布曲线类似[4]。

- RESL成本投入%

  预算中划分给架构设计以及风险应对的部分所占的百分比。计算方法是用RESL的时间投入百分比乘以每个RESL等级的项目平均人力投入。举例来说，对于“非常低”的RESL投入级别来说，就是5 × 0.3=1.5。

- 投入增量

  即当前等级的RESL投入成本百分数减去前一等级的投入成本百分数。例如，对于“低”等级的RESL投入来说，投入增量就是4–1.5=2.5%。

- 返工工作量的规模增长指数

  从161个项目的评级情况来看，RESL参数对于软件项目的工作量有着指数级的影响。

  表格其余部分显示了四个不同的系统规模（10 KSLOC到10000 KSLOC）的数值，以及5种不同的RESL级别下的ROI。计算的顺序如下：

- 附加的工作量

  将返工的规模增长指数（如1.0707）乘以系统规模（如10 KSLOC），并计算增加了多少工作量。比如说，对于10 KSLOC的项目来说，“非常低”的RESL水平的附加工作量计算如下：

$$ROI = \frac{(B-C)}{C}$$

- 效益增量

  前一评级下的附加工作量减去当前评级下的附加工作量。比如，对于“低”的RESL水平来说，效益增量就是17.7–13.9=3.8。

- 成本增量

  与前面说明过的投入增量相同。

- ROI增量

  利益减去成本再除以成本：

$$= \frac{10^{1.0707} - 10}{10} \times 100$$
$$= 17.7$$

  比如对于1万行代码的项目来说，“低”RESL级别的ROI增量计算如下：

$$= \frac{(3.8 - 2.5)}{2.5}$$
$$= 0.52$$

这些计算结果显示，随着RESL的投入逐渐增大，其所带来的ROI增量也越来越小。较大的项目的ROI增加比较多，增加的持续时间也比较长，因为这些项目常常一开始返工水平就很高，使得其进步的空间也更大。图10-9展示了这些结果。

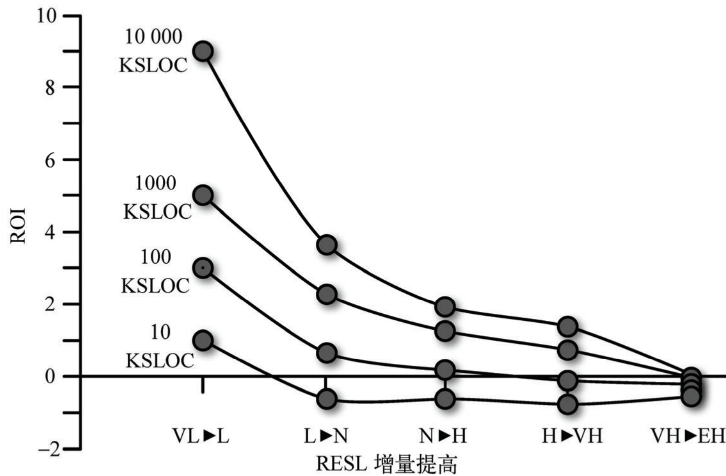


图10-9　系统架构设计带来的ROI增量

10.4　那么到底架构要做到什么程度才够

现在，我们可以使用前面的计算结果，来大致回答这个项目规划阶段的核心问题。

我们进行这个研究的动机源自一个非常大的软件项目（超过10000 KSLOC），项目组希望承包开发的公司能够尽快地开始工作，所以他们冒着架构设计以及风险应对不足的风险，没有把计划和规格文档放到征求建议书（RFP）中。这个项目组必须找到它的“平衡点”：一方面必须避免承包商做出不合格的组件，因为那样就意味着返工，也就会造成日程超标；另一方面，他们又必须考虑到系统架构设计以及风险应对的时间成本问题，因为如果前期时间花费过多可能造成承包商没有足够的时间来进行开发。本节将展示的就是如何用COCOMO II的RESL等级来找出适合这个项目以及其他规模项目的架构设计平衡点。

表10-4展示了在10 KSLOC、100 KSLOC以及10000 KSLOC的项目中，每个RESL等级所对应的架构设计投入百分比。表中还展示了相应的总交付延迟百分比，这些百分比的计算方法是用架构设计的时间投入加上返工消耗的时间。此处为了把附加的工作量表示为附加的时间安排，我们假设返工期间团队的规模不变。表10-4的最后两行显示，当架构投入在33%或者更少时，这些附加投入比节省的返工时间要多（10000 KSLOC的项目），而超过33%以后，总延迟百分比增加了。简而言之，就是对于10000 KSLOC的项目来说，延迟最小的架构设计投入百分比的平衡点是33%。

图10-10上半部分用图形的方式展示了表10-4中的数据。图表显示，对于10000 KSLOC的项目来说，平衡点是在架构投入37%前后的扁平区域（这样的分析结果使得这个项目的项目组额外增加了18个月来做架构设计）。对于100 KSLOC的项目来说，平衡点大概在20%前后的扁平区域。对于10 KSLOC的项目来说，平衡点大概在5%前后。图10-10和表10-4都表明，对于小项目来说，架构设计投入的用处不大，但是项目越大，其重要性和必要性都越来越明显。

表10-4　架构设计投入对项目延迟的影响

COCOMO II RESL等级	非　常　低	低	中	高	非　常　高	极　　高
RESL时间投入%	5	10	17	25	33	>40（50）
返工工作量的规模增长指数	1.0707	1.0565	1.0424	1.0283	1.0141	1.0
10 KSLOC的项目						
附加的工作量%	17.7	13.9	10.3	6.7	3.3	0
项目延迟%	23	24	27	32	36	50
100 KSLOC的项目						
附加的工作量%	38.4	29.7	21.6	13.9	6.7	0
项目延迟%	43	40	38	39	40	50
10000 KSLOC的项目						
附加的工作量%	91.8	68.3	47.8	29.8	13.9	0
项目延迟%	96	78	65	55	47	50

图10-10的下半部分展示的是最近我们对COCOMO II数据库中项目的需求变动性及系统危险性的进一步分析。图中的实线表示100 KSLOC的项目的任务平均返工以及架构设计的成本，左边写着“总成本”的几条线代表了总成本。虚线表示的是由于需求变动程度比正常的大而造成架构设计成本增加50%的时候，架构设计及总成本关系的变化。定量地说，在这种情况下的高回报率的架构投入平衡点就从20%转移到了10%（实际上应该是15%，因为巨大的变动程度带来了50%的额外成本）。这样看来，对于需求变动非常大的项目来说，由于进行各种分析的成本较高，而在需求迅速变化的情况下文档又必须不停整改，所以在架构设计、可行性分析以及其他文档上的高投入会导致入不敷出。

图10-10的下半部分中的短划线表示的是对未识别的架构不足而引起的系统错误的外部成本所进行的保守分析。这些由架构设计不足所导致的损失不止包括额外的返工时间，还包括了组织在效率和生产力上造成的损失。保守估计，此类成本将导致项目返工成本提高50%。在大多情况下，对于质量要求较高的系统，追加的成本会相应地更高。

定量地说，对于100 KSLOC的项目，高回报率的架构设计投入平衡点就从约20%转到了超过30%。正如前面所说的，所有的这些平衡点其实都是相对扁平的“平衡区域”，即平衡点前后的5%～10%的范围。不过，如果选择的投入度比较靠向平衡区域的边缘部分而项目的预期又过于乐观的话，遭遇重大损失的风险就会增加。

如果总结一下我们从图10-10中学到的经验，那就是项目的规模、风险和稳定性越大，那么对架构设计和可行性评估的需求就越大。不过，对于规模非常小、风险很低而且需求变化还很快的项目来说，架构设计做多做少并没有什么区别。并且，对于这种项目来说，如果不停反复地做架构设计，就有可能让ROI变成负数。在这类项目中，敏捷的方法（如XP）会更有效。总的来说，有了以证据为基础的详述和计划并不能保证项目一定会成功，但是通常来说可以解决现在很多项目中都存在的预算超时及不足的问题。

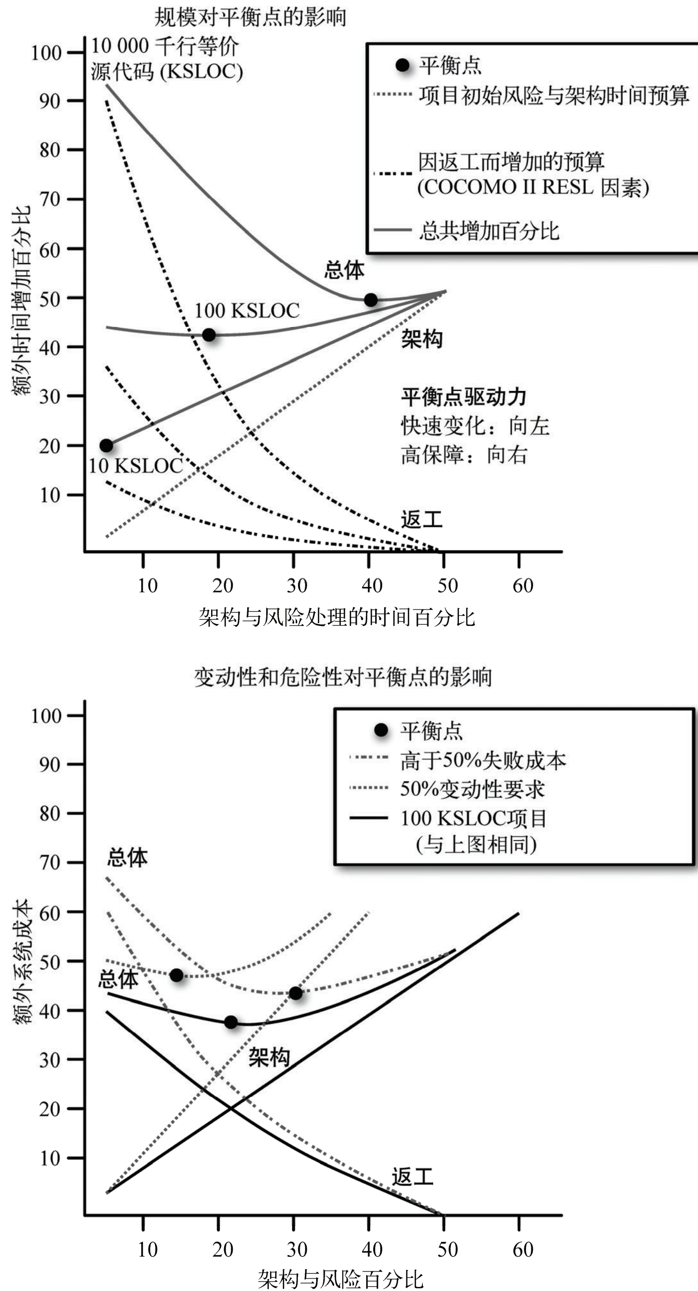


图10-10　项目的架构设计不足所引发的额外成本及其相应的架构设计投入平衡点

我们还有一个最后的忠告，那就是图10-10中所展示的平衡点并不一定适合所有的项目，它们只能作为前期指标来找出适合的架构设计投入的范围。当项目进行到了详细的规划及预算制定

阶段时，其他的一些影响成本的因素，如个人能力和经验、程序的复杂度、旧系统的迁移复杂度、技术风险、流程的成熟度以及支持工具等，也可能会对架构设计投入的平衡点造成影响。最近，研究人员们设计并标准化了一个新的模型，称之为建设性系统工程模型（Constructive Systems Engineering Model，COSYSMO），这个模型可以更精准地确定项目所需要的架构设计投入[32][33]。

10.5　架构设计是否必须提前做好

很多敏捷项目都通过进行大量的架构设计工作获得了成功，但是这些项目都把架构设计工作通过重构的方式分散到了整个项目周期。只要项目规模小、风险不高，这样做是很高效的，但是如果项目的规模和风险变大，那么就有可能造成“避重就轻”式的问题设计。

比如在[17]中描述的那个租约管理器项目，当这个项目以一个较小的规模逐渐成长壮大的时候，旧有的依靠开发人员之间的默契来开发以及就地解决的重构工作做法却无法跟上其成长的脚步。还有一类问题，就是当选择基于商用的现成产品进行开发时，我们会迅速地开发并尽快满足客户的需求，但是后期却发现这些现成的产品无法适应规模的增长，无法处理增加的工作量，而这些产品又没有源代码可供重构。有个与此相关的问题，就是有时候项目组会推后一个贯穿整个程序的功能，比如安全检查，但是在一年半载之后，却发现此时想要再添加这个功能或者把这个功能重构进早前的架构设计已经不可能了。所以，我们一定要预先考虑到需求的发展，并将这些考量提前加入到架构的设计中。

10.6　总结

根据我们40年间对软件的问题修正的成本和延迟的研究数据，大型项目中，在产品发布后修正问题和在需求定义阶段修正问题的成本比率一直都保持在100：1左右。但是，通过在项目早期投入更多时间和人力对需求进行确认和验证，这个比率可以得到显著地降低。正如图10-6中CCPDS-R项目的数据所显示的那样，如果高风险的修复早些解决，这个比例可以接近1：1。

小型项目的比率则保持在5：1左右。虽然比率不高，但是我们仍然可以将其再降低，方法就是启用优秀的人才以及敏捷开发方法，比如结对编程和持续集成，这样就能缩短修正延迟的时间。小型、低风险度的项目还可以把架构设计通过重构的方式分散到整个开发流程中，但是需要小心不要用“避重就轻”的方式来做早期的架构设计，以免后期无法通过重构的方式来加入需要的功能，比如使用扩展度差的现成产品或者安全性低的数据和控制结构等。

最近的一些关于架构设计以及风险应对的证据（如CCPDS-R相关的证据）表明，要找出所谓“架构设计是否足够”的最高回报率的平衡点到底是多少，不仅需要考虑项目的规模及重要性的问题（项目越大、越重要需要的架构设计投入就越多），还需要考虑需求的变动性（需求变化快的项目可能会导致大量的文档修订、修改工作而使得项目进度变慢）。想要进行更加详细的项目规划和预算制定，我们还必须按照其他对成本造成影响的项目、人员和产品因素对平衡点进行调整。最近研究出的一个COSYSMO模型可以帮助我们做这样的调整。

非常大型的项目比较可能会有高重要性且高稳定性的元素（比如安全性相关的元素），还可能

会有需求变化度高的元素（如用户界面、外部系统界面、设备驱动程序、数据库表结构等）。在这样的情况下，我们可以使用混合式方法：用敏捷方法来开发变化快的部分，而使用基于计划的方法来开发较为稳定和重要的部分。这种方法有一个前提，那就是系统整体是基于Parnas报告[24]中所描述的用模块来封装变动源的信息隐藏架构。

所以，对于未来那些需求变化更快的项目来说，没有放之四海而皆准的解决方案。对于既包含了重要元素又多变元素的大型或者企业级的项目来说，最好采用风险驱动的流程生成法，比如增量保障模型（ICM）或者风险驱动版的合理统一流程。这些模型生成法使用开发人员提交的项目可行性数据来确定项目的风险。对于那些不得不面对不断增长的规模、重要性、变动性和复杂性的未来项目来说，想要成功就必须依赖这些数据。此外，使用这种基于证据的模型还有个双重好处：一方面可以降低风险，另一方面可以充实证据知识库，以便于将来进行分析研究，找出更多改善项目和企业的方法。

10.7　参考文献

[1] [Beck 1999] Beck, K. 1999. *Extreme Programming Explained*. Boston: Addison-Wesley.

[2] [Boehm 1976] Boehm, B. 1976. Software engineering. *IEEE Transactions on Computers* 25(12): 1226-1241.

[3] [Boehm 1980] Boehm, B. 1980. Developing small-scale software application products: Some experimental results. *Proceedings of the IFIP Congress 1980*: 321-326.

[4] [Boehm 1981] Boehm, B. 1981. *Software Engineering Economics*. Upper Saddle River, NJ: Prentice-Hall.

[5] [Boehm 1996] Boehm, B. 1996. Anchoring the software process. *Software* 13(4): 73-82.

[6] [Boehm 2000] Boehm, B. 2000. Unifying software engineering and systems engineering. *Computer* 33(3): 114-116.

[7] [Boehm and Lane 2007] Boehm, B., and J. Lane. 2007. Using the incremental commitment model to integrate system acquisition, systems engineering, and software engineering. *CrossTalk* (Oct. 2007): 4-9.

[8] [Boehm and Lane 2009] Boehm, B., J. Lane. 2009. Guide for Using the Incremental Commitment Model (ICM) for Systems Engineering of DoD Projects, Version 0.5. USCCSSE-2009-500. Available at *http://csse.usc.edu/csse/TECHRPTS/2009/usc-csse-2009-500/usc-csse-2009-500.pdf*.

[9] [Boehm and Port 1999] Boehm, B., and D. Port. 1999. Escaping the software tar pit: Model clashes and how to avoid them. *ACM Software Engineering Notes* 24(1): 36-48.

[10] [Boehm and Port 2001] Boehm, B., and D. Port. 2001. Balancing discipline and flexibility with the spiral model and MBASE. *CrossTalk* (Dec. 2001): 23-28.

[11] [Boehm and Royce 1989] Boehm, B., and W. Royce. 1989. Ada COCOMO and the Ada process model. Paper presented at the 5th COCOMO Users' Group, October 17-19, in Pittsburgh, PA. Available at *http://csse.usc.edu/csse/TECHRPTS/1989/usccse89-503/usccse89-503.pdf*.

[12] [Boehm et al. 1998] Boehm, B.A., Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy. 1998. Using the WinWin spiral model: a case study. *IEEE Computer* 31(7): 33-44.

[13] [Boehm et al. 2000] Boehm, B., C. Abts, A.W. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. 2000. *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice-Hall.

[14] [Boehm et al. 2010] Boehm, B., J. Lane, S. Koolmanojwong, and R. Turner. 2010. Architected Agile Solutions for Software-Reliant Systems. In *Agile Software Development: Current Research and Future Directions*, ed. T.

Dingsøtr, T. Dybå, and N.B. Moe, 165-184. Berlin: Springer-Verlag.

[15] [Clements et al. 2002] Clements, P., R. Kazman, and M. Klein. 2002. *Evaluating Software Architectures*. Boston: Addison-Wesley.

[16] [Daly 1977] Daly, E. 1977. Management of software engineering. *IEEE Transactions on Software Engineering.* 3(3): 229-242.

[17] [Elssamadisy and Schalliol 2002] Elssamadisy, A., and G. Schalliol. 2002. Recognizing and Responding to “Bad Smells” in Extreme Programming. *Proceedings of the 24th International Conference on Software Engineering*: 617-622.

[18] [Fagan 1976] Fagan, M. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15(3): 182-211.

[19] [Jacobson et al. 1999] Jacobson, I., G. Booch, and J. Rumbaugh. 1999. *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

[20] [Kruchten 1999] Kruchten, P. 1999. *The Rational Unified Process: An Introduction*. Reading, MA: Addison-Wesley.

[21] [Li and Alshayeb 2002] Li, W., and M. Alshayeb. 2002. An Empirical Study of XP Effort. Paper presented at the 17th International Forum on COCOMO and Software Cost Modeling, October 22-25, in Los Angeles, CA.

[22] [Maranzano et al. 2005] Maranzano, J., S.A. Rozsypal, G.H. Zimmerman, G.W. Warnken, P.E. Wirth, and D.M. Weiss. 2005. Architecture reviews: Practice and experience. *Software* 22(2): 34-43.

[23] [McGibbon 1996] McGibbon, T. 1996. Software reliability data summary. Data & Analysis Center for Software Technical Report.

[24] [Parnas 1979] Parnas, D. 1979. Designing software for ease of extension and contraction, *IEEE Transactions on Software Engineering* 5(2): 128-138.

[25] [Pew and Mavor 2007] Pew, R., and A. Mavor, ed. 2007. *Human-System Integration in the System Development Process*. Washington, D.C.: National Academies Press.

[26] [Rechtin 1991] Rechtin, E. 1991. *Systems Architecting*. Englewood Cliffs, NJ: Prentice-Hall.

[27] [Rechtin and Maier 1997] Rechtin, E., and M. Maier. 1997. *The Art of Systems Architecting*. Boca Raton, FL: CRC Press.

[28] [Reifer 2002] Reifer, D. 2002. How to get the most out of extreme programming/agile methods. *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods*: 185-196.

[29] [Royce 1998] Royce, W. 1998. *Software Project Management: A Unified Framework*. Reading, MA: Addison-Wesley.

[30] [Shull et al. 2002] Shull, F., V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoreiro, and M. Zelkowitz. 2002. What we have learned about fighting defects. *Proceedings of the 8th International Symposium on Software Metrics*: 249-258.

[31] [Stephenson 1976] Stephenson, W. 1976. An analysis of the resources used in the SAFEGUARD software system development. *Proceedings of the 2nd international conference on software engineering*: 312-321.

[32] [Valerdi 2005] Valerdi, R. 2005. The constructive systems engineering cost model (COSYSMO). PhD diss., University of Southern California.

[33] [Valerdi and Boehm 2010] Valerdi, R., and B. Boehm. 2010. COSYSMO: A systems engineering cost model. *Génie Logiciel* 92: 2-6.

第11章

康 威 推 论

Christian Bird

设计和编程都是由人来进行的；忘掉这一点的话，其他任何事都没有意义。

——Bjarne Stroustrop

11.1 康威定律

1967年，一位名叫梅尔文·康威（Melvin Conway）的早期计算机科学家、程序员及黑客向《哈佛商业评论》提交了一篇名为“组织是如何进行发明创造的？”的文章，但是立刻被退稿了，理由是他“没能证明该理论”。

对于软件设计界来说，幸运的是这篇文章随后被*Datamation*杂志（当时一流的IT杂志）接受，并于1968年发表[4]。后来，Fred Brooks在其开创性作品《人月神话》[2]中，将康威的某些言论归纳为“康威定律”，这个名字就这样流传了下来。虽然康威的言论在当时没有得到实证研究的支持，但是近年来这条“定律”却吸引了不少人的关注。本章包含一些支持康威定律的证据，毫无疑问这些证据如果出现在1967年的话，会帮上康威的大忙。而在今天，这些证据的价值则在于它们证明了“康威定律”是现代软件开发人员的试金石。

康威的文章这样描述这个理论：

> 任何组织设计出来的系统（此处指广义的系统）的结构都会照搬这个组织的沟通结构。

康威引用了一些坊间的传闻来作证：某个合同研究组织需要编写一个COBOL和一个ALGOL编译器，可供调遣的程序员共有8人。在对难度和时间做了初步的估计之后，5个人被指派去写COBOL编译器，3个人写ALGOL编译器。结果，COBOL编译器的运行分5个步骤，ALGOL编译器的运行分3个步骤。

虽然看上去既简单又不能作为有效的证据，但是这个现实世界中的例子却让人印象深刻，让人看到了开发团队的组织结构对于软件设计和架构的影响。

康威说《哈佛商业评论》拒绝发表这篇文章，“并非针对这篇文章，更多地表明大家对于‘证明’二字有着不同的理解”。

从1967年算起，软件工程的实证研究已经得到了长足的发展，这一章将展示一些重要的研究，这些研究都评估了康威定律的有效性。很多实证研究都有重大的缺陷，没有一个是完美，但是这些研究都是按照成熟的指导方针来实施的，所以我们有理由把这些研究的成果作为有力的证据，证明将软件结构对应到组织结构不但是一种趋势，而且还非常有*必要*。

任何大型的系统都是由稍小一些的*子系统*相互连接组成的。这些子系统又可以被看成是由更小的模块组成的。系统的设计是由模块组成的，每个模块完成一部分的功能，并连接一部分其他的模块。软件开发和其他学科一样，使用*接口*（interface）这个术语来描述模块间用于相互连接的机制，或者用于公开模块所提供的功能。也就是说，系统可以被描述为由节点（模块/子系统）和边（以接口形式存在的相互连接）组成的图。

根据系统的设计起源，康威提出了自己的“证明”，即系统的结构正好反映了组织的结构。对于系统中的任意一个模块（假设为x），我们都能够在组织中识别出设计它的那群人（我们称这群人为X）。也就是说，系统中的每个模块都有一个与之对应的设计团体。不过，需要注意的是两者之间并不一定是一一对应的关系，因为同一个设计团体可能设计多个模块。由两个不同的设计团体（假设为X和Y）分别设计的模块或者子系统（假设为x和y）之间可能需要做连接，也可能不需要。如果需要做连接，那么X和Y团体必然需要商量并确定一套接口规范，让两个模块可以进行沟通。如果不需要做连接，那么两个设计团体就无需商量，而X和Y之间也就没有沟通。

也就是说，至少在开始的时候，系统结构以及创建它的组织的结构之间是有紧密关联的。不过，软件与开发维护软件的组织一样，是不断变化的：需求会变化，团队之间的人员会有调动，设计也会有改变。

虽然康威的理论关注于软件的初始设计阶段，但是很多研究人员发现，这个现象在后期阶段仍然存在。所以，康威定律就有了一个可能的推论：

> *系统的结构能和开发组织的结构高度对应的软件比那些不能对应的软件要“更好”（广义地说）。*

“更好”一词的定义根据项目的背景而定。大部分软件开发组织都有两个高层次的目标：保持高效率，同时保证软件质量。本章展示了一项关于康威定律和开发人员完成任务的速度（即效率）之间的关系的研究，以及另一项关于康威定律如何影响发布后的故障（即质量）的研究。两者的结果都支持康威的判断。我们还将展示一项对5个著名的开源软件的研究，研究结果显示，在任其自由发展的情况下，一个成功的软件项目（至少在开源软件这个领域的项目）的沟通结构常常会和软件的结构对应起来而变得有些模块化。

对于本文中包含的每项研究，我们都将指出其背景、使用的方法、研究结果以及相关结论。此外，我们还将根据我们的结论来推荐一些实用的方法对软件项目进行改善。特别是，我们认为不应该让程序员的任务分配来决定软件结构，因为这样很危险。我们应该首先确定我们需要的软件结构，再相应地调整我们的组织结构。

我们支持实证主义的观点，即证明一个理论是很困难的，但是在我们每次尝试推翻它而不能成功的时候，我们对它就更多一点信心。所以，我们希望本章能增加读者对康威定律及其推论的信心。

11.2　协调工作、和谐度和效率

在2006年的时候，Marcelo Cataldo等人[3]提出了一个问题："在现实世界的大型软件项目中，协调工作对效率有多大影响？"这个问题的意义很明确：如果协调工作对效率的影响很大，那么软件开发组织就应该竭尽全力地保证协调工作的进行。如果协调工作的影响不大，组织就应该把时间和金钱投资到项目的其他方面，如更多的人手、更好的设备或者把办公地点搬到经济更发达的地方等。

为了回答这个问题，他们研究了一家大型公司的一个重要项目中的工作依赖性、协调需求以及完成开发任务所需要的时间。这个项目用修改请求（Modification Request，MR）来代表一个任务。MR代表着对系统的一切修改请求，包括添加新功能、修复问题以及改善代码库的维护任务。完成MR任务的团队成员常常需要和他人协调进行修改，其主要原因有二。

- 有的MR需要依赖其他MR。比如说，如果某个MR请求为文字处理器加入检查拼写的功能，那它就必须等到添加字典功能的MR完成之后才能进行。
- 软件项目几乎总是由一些相互依赖的模块组成。因此，对于系统做出的修改很少是孤立的，这些修改常常会影响到"周边"的模块。

基于这些原因，Cataldo等人认为，当任务的依赖关系可以和协调工作可以"相互适应"时，完成任务就会更容易。他们把这种适应性称之为*和谐度*，并从大型软件项目中采集了数据以评估他们的假说。虽然看上去似乎比较简单，但是如何衡量软件团队中的协调活动却不是那么容易就能解决的问题。实证研究的困难有一半都是在于找到一个好的衡量方法，这个方法必须可以基于当前可用的数据来进行衡量。Cataldo等人制定了4个衡量方法，以便于从MR中找出软件开发者的协调和谐度信息。这些方法除了分析团队之间的直接沟通，还会分析团队中有利于沟通的特征。可以直接观察到的沟通包括MR中的注释信息，以及互联网中继聊天（IRC）中的内容。沟通的可能性则由团队成员的地理距离以及组织内的职位决定。四个和谐度指标如下。

- 结构和谐度

  这个指标代表着同一个团队中开发人员的协调是否方便。我们有足够的理由相信，团队内的沟通比团队间的沟通更容易。团队成员内部沟通一般通过会议或者其他工作相关的活动。他们通常相互认识并且曾经沟通过。此外，正是因为管理者认为开发人员的工作需要相互合作，才会把他们放到一个团队里的。

- 地理和谐度

  这个指标把开发人员之间的物理距离看做是沟通难度的代表。当两个开发人员在同一栋大楼工作时，他们要联系对方就更加容易，无需预约，直接面谈或者随时开个小会都可以。而且在同一个地点工作的人之间也不存在时差的问题。此外，由于沟通是面对面的，所以沟通层面也更加丰富，包括面部表情等暗示都会成为沟通的一部分。他们可以在白板上勾勒出自己的想法和计划，或者一起坐在屏幕前看代码，这些沟通方式对于距离远一些的人们来说就比较难了。

- 任务和谐度

  这个指标衡量在MR注释中的沟通。在Cataldo所研究的软件项目中，开发人员能够在在线MR追踪系统中为MR写注释以便让其他人了解这个MR。这样，对某个MR感兴趣的开发人员之间就可以进行沟通。开发人员用MR注释或者评论的方法来沟通，不但代表着某种程度的协调，还意味着他们交换了与该MR相关的技术信息。

- IRC沟通和谐度

  这个指标测量开发人员使用IRC（也就是即时通信）进行的沟通。虽然IRC不如面对面的沟通那么丰富，但是它作为一种同步沟通手段，可以帮助开发人员们相互提问或者为已经协调好的修改任务制订计划。除了他本人以外，Cataldo额外邀请了两个评估人员来阅读IRC对话记录，并手动把它们对应到具体的MR。为了确保评估标准的统一性，三个评估者同时对10%的MR进行了对应，而其结果的相同程度高达97.5%，也就是说IRC到MR的对应是比较准确的。

这个软件项目包括144名开发人员，他们分别处在8个团队，而这些团队又分布在3个不同的位置。数据收集涵盖了3年时间和4个发行版本。这一研究方法的关键在于观察每个开发人员分配到的任务，以及每个任务的依赖关系。如果张三正在进行任务t1，而这个任务需要依赖任务t2，而t2又是由李四来做的，那么张三和李四就可能需要进行协调。Cataldo等人使用的方法是：找出MR是谁完成的，以及在完成MR时修改了哪些文件。也就是说，开发人员的任务就是一组文件。通常来说，完成一个MR需要修改多个文件。然后，计算有多少次是同时修改两个文件来完成一个MR的，就能得出两个文件之间的相互依赖度。具体来说，就是如果文件f1和f2在过去曾经多次被同时修改来完成MR，并且张三需要修改f1来完成他的MR而李四又需要修改f2来完成自己的MR，那么对他们来说沟通可能就很有必要了。

当需要完成MR时，我们可以用这些规律来确定我们预期会发生的协调工作。而Cataldo使用了之前介绍的四个方法测出了开发人员之间的实际协调规律。如果预期和实际发生的协调工作紧密吻合，那么和谐度就高，反之如果预期的协调没有发生，那么和谐度就低。他们使用这个方法来计算了每个MR的每种四种和谐度，然后调查了每个MR的每种和谐度及其完成时间之间的关系。

四种形式的协调代表了整个开发组织的沟通机制。可以从一个比较高的角度来看待Cataldo等人问的这个问题，即："如果软件结构和组织的沟通结构相对应的话，任务是不是就完成得更快？"如果答案是"是"，那么我们就可以把这个答案作为证据，证明遵守康威定律会带来更高的效率，也会从完成任务时间的角度证明，康威的推论确实是正确的。

在研究中，Cataldo等人还严密控制了一些会影响MR完成时间的因素。比如，必须等到很多其他的MR完成之后才能进行的MR的完成时间会长一些，而优先级较高的MR一般来说会完成得更快。总的来说，他们对以下这些因素进行了控制：依赖关系、优先级、工作调动、需要修改的文件数量、MR的来源（内部或客户提出）、负责MR的开发人员的经验（如编程的经验、进入公司的时间以及在该MR相关的组件上工作的时间）、MR所在的发行版本以及开发人员的工作负荷（即同时指派的其他MR的数量）等。这样，就可以避免直接对比实际情况差距很大的两个MR（比

如初始开发阶段、由新手执行的低优先级MR以及一个维护阶段、由老手执行的高优先级MR）的执行时间。如果不对比这些背景情况，那么最后得出的结论就可能是“协调工作不到位延长了MR的完成时间”，而真正的原因却可能是经验不足或者项目所处的阶段问题。这种严谨的态度和对细节的注意让这项研究的结果更好地反应了现实世界的情况，而不仅仅是一堆靠不住的统计数据。

在经过仔细的量化分析之后（具体细节此处不表，有兴趣的读者请阅读该论文的完整版），研究人员们发现，在背景情况相似的前提下，高和谐度（或者说“协调适合度”）的MR所需要的时间比低和谐度的少。

有趣的是，*结构和谐度*（即和团队结构有关的和谐度）对于减少MR所需时间的贡献最大，其次是*地理和谐度*（即需要合作的开发人员的距离）。

另一个重要的发现是版本越靠后，IRC*和谐度*影响越大。图11-1显示，软件的架构可能会随着时间的推进而产生改变，*结构和谐度*的下降意味着需要多个团队才能合作解决的MR增多。而由于IRC沟通可以降低跨团队协调的难度，所以，IRC和谐度变强也有可能意味着开发人员在项目后期可以更熟练地使用IRC来和不同团队及工作地点的同事协调工作。

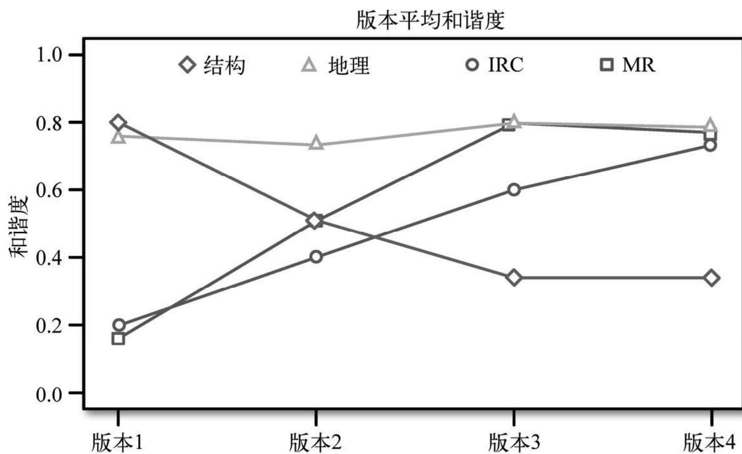


图11-1　每个版本的平均和谐度

给我们的启示

我们已经知道了很多，但是这些对于你的项目来说有什么意义？对于个案研究来说，重要的是找出其普遍适用性。如果要将研究的结论推广到其他项目，那么这些项目和研究对象就必须是相似的。幸运的是，Cataldo等人研究的软件项目非常庞大，共有超过100名开发人员以及成千上万的文件和任务（即MR）。这个项目由分布在不同工作地点的多个团队共同进行开发，研究的时间段也从初始版本一直延续到后续几个版本。所以，我们对于该研究结果在软件开发中的普遍适用性还是比较有信心的。

虽然你无法预知软件会出现哪些bug或者需要哪些新功能，但是在软件的设计阶段，还是应该对可能产生紧密依赖关系的部分有所认识。这种存在于组件之间的紧密依赖关系意味着如果协调没做好，那么系统的这些部分就可能会受到较大的影响。我们在为开发人员及团队进行任务分配的时候，应该尽量做到方便参与开发的各方进行沟通，尤其是相互依赖的组件的开发人员之间的沟通。也就是说，我们最好在设计完成之后再确定团队的结构，或者至少保证团队的结构有一定的弹性。Cataldo等人注意到结构和谐度会随着时间的推移而下降，这表明团队结构和代码依赖关系之间的对应关系改变了。在版本交替的时候重新组织团队，这样就能提高结构和谐度，并缩短MR的解决时间。同样，分配任务时我们还应该考虑地理位置的因素。我们应该尽量将相互依赖度较高的组件交给同一个工作地点的开发人员或者团队，这样他们就可以用更多的方式来进行协调。

MR和IRC和谐度则更加地动态，会随着项目的推进而改变。有一点很明确，那就是在解决某个MR的时候用注释进行协调的开发人员越多，解决的速度就会越快。我们应该宣传和鼓励这样的沟通方式。当前的研究课题主要是使用工具来帮助开发人员及管理者找出缺少这类协调的任务，并确定相应组织协调工作的人员。此外，开发组织应该积极采取更好的沟通工具，以便于进行跨地域和跨组织的协调工作。在项目开始之前就对这类沟通工具和技术进行投资，那么受制于协调需求的任务就会减少，效率也能得到提高。

11.3 微软公司的组织复杂度

我们探讨了康威推论对于开发人员效率的影响，并分析了如何使用“和谐度”（Cataldo创造的术语，即符合康威推论的程度）来帮助程序员更快地完成任务。对于任何软件开发项目来说，另一个不能忽视的重点是质量。我们都知道，发现缺陷的时间越晚，修复成本就越高。在软件已经正式发布之后发现的缺陷会带来巨大的损失，所以我们必须将这种可能最小化。这样的缺陷所带来的损失并不一定只是金钱上的，当用户在软件中发现bug的时候，他们对于产品整体质量和公司名誉的看法都会下降，会对公司市场地位造成无法磨灭的影响。

因为缺陷的相关成本很高，而康威又下过这样的断言，所以Nachi Nagappan、Brendan Murphy和Vic Basili三人对Windows Vista的组织结构和发布后的缺陷进行了深入的研究[5]。研究发现，两者之间有着惊人的强烈关联。他们发现实际上对组织结构的度量才是*决定软件质量的关键因素*，比软件本身的任何特质都要重要。举个简单的例子，就Windows Vista而言，如果你想知道某个软件组件是否存在bug，你可以不去看代码，直接去看编写这个组件的开发人员的组织方式就行了。

他们研究的核心前提是：软件开发是一项需要集体努力的工作，而组织层面的复杂性会让协调工作变得困难。对于实证研究来说，都必须把具体的现象抽丝剥茧，变成可衡量的东西。不过，衡量软件的质量还是很容易的。Windows Vista包含成千上万个“二进制文件”，即编译好的代码。这些文件包括可执行文件（.exe）、共享库文件（.dll）以及驱动程序（.sys）。一旦某个组件出现故障，系统就会分析出造成故障的二进制文件，并发送错误报告到微软，（记得当程序崩溃时出现的搞笑对话框吗？）然后微软可以用这些报告来确定bug修复的优先级。他们利用这些信息，

与其他的bug报告和相关的修正数据，来判断每个二进制文件出错的可能性。为了评估康威推论的效果，Nagappan把Vista中的每个二进制文件都分为两种，即容易出错或者不容易出错，分类基于二进制文件中不同错误的数量。需要注意的是即便有成千上万个用户发送崩溃报告，只要是这些崩溃都是由一个缺陷引起的，就只算作一个缺陷。也就是说，出错倾向是用于衡量某个二进制文件中的已知缺陷数量的指标。

测量组织复杂度要稍微困难些。Nagappan、Murphy和Basili提出了组织复杂度的8个度量法，并研究了各个度量法和错误倾向之间的相关性，并在控制其他指标效果的前提下，利用了回归模型来分析对比各个指标的效果。他们使用了版本控制系统中的数据来确定哪些开发人员修改了源文件。从编译信息中可以找出用于编译每个二进制文件的源代码文件。为简洁起见，如果某个开发人员修改了某个二进制文件对应的源代码文件，那就认为他修改或者插手了这个二进制文件。

Nagappan、Murphy和Basili还从公司的结构图中搜集数据，以便识别开发人员的组织关系。这套度量法所使用的信息都是来自上述来源，没有使用其他信息。这8个组织结构指标如下所示。

- 开发人员的数量

  这个指标指的是修改了某个二进制文件并且仍在在本公司任职的开发人员数量。我们预计这个指标越高，二进制文件就越容易出错。我们的这个判断是基于Brooks的结论[2]，即如果某个组件的开发人员有n个，那么他们之间就存在$n(n-1)/2$条可能的沟通渠道。当软件团队的规模呈线性增长时，沟通成本却*不会只是线性*增长，而是要快得多。当沟通的渠道越来越多，那么就有可能带来各种问题，比如协调的问题，设计和实现的不匹配问题，破坏别人代码的问题以及对设计初衷的误解等。所以，我们认为二进制文件对应的开发人员越多，那么错误也就越多。

- 离职开发人员的数量

  这个指标指的是修改了某个二进制文件但又在正式发布之前离开了公司的开发人员数量。我们认为这个指标越高，问题就越多。这个指标评估的是所需转移的知识量。当某个二进制文件的开发人员离职的时候，另一个对这些代码经验较少的开发人员就可能会顶替他的工作。新的开发人员很可能不像原来的那位一样了解组件设计、关于bug修复的各种考量以及这些代码所涉及的其他人员。这就会增加出错的可能性，以及引入缺陷的可能性。

- 修改频率

  修改频率是二进制文件被修改的次数，无论每次修改了多少行都只算一次。也就是说，每次修改都是版本控制系统中的一个独立的版本提交。高修改频率被认为是代码质量差的表现。如果某个二进制文件修改太多次，那么就有可能意味着代码的稳定性或者控制性不强，即便这些修改都是由某一小群开发人员做的。我们可以把这个指标和开发人员数量以及离职开发人员数量相结合，这样就能看到更为全面的修改分布情况。比如，是不是大部分修改都是由某一个开发人员做的，还是说这些修改者的分布非常广泛？把开发人员和修改数量关联起来，可以避免这样一种情况，那就是少数的开发人员做了几乎所有的修改。在这种情况下，指标就会出问题，更会导致错误的结论。

- 主负责人深度（Depth of Master Ownership，DMO）

  这个指标评估的是某个二进制文件的相关任务在组织内的分布有多广泛。对于任意一个二进制文件，组织内的每个人都有一个对应的修改次数，这个次数包括他对这个文件做出的修改次数，再加上他的每个下级对这个文件的修改次数。换言之，每个开发人员所做的修改次数会在他们的管理者那里汇总，并加上管理者自己的修改次数。DMO是指修改次数占总数75%以上的人（即该二进制文件的主负责人）在组织内的最低级别。这个指标越高，组织中的主负责人等级就越低（即越深）。我们认为，这个指标越高，故障就会越少。对于某个二进制文件来说，如果大部分的修改都是由一群组织等级相近、从属于同一个管理者的开发人员负责的，那么这个较为基层的管理者就对这个二进制文件负责，而DMO也就会很高。这就意味着某个二进制文件的相关开发人员都联系紧密，并且协调工作无需组织内高层参与。如果从比较高级别管理者中才能找出负责75%的提交的人，那么谁才是负责人就不那么明显了，而修改也是由组织内多个不同的部分做出的。这就可能造成决策方面的问题，因为有多方参与修改，而每方的目标又有不同。而由于高层管理者要管理的人很多，常常无法参与大部分的实际工作，沟通就常常会被延迟、取消或者造成误解。

- 组织的开发参与百分比

  这个指标指的是回报给主负责人（见前一个指标）的开发人员所占组织总人数的比例。如果组织中有100个人在开发某个二进制文件，其中25个人直接汇报给主负责人，那么这个指标的值就是0.25。我们认为，这个指标的值越低，就意味着二进制文件的错误率越低。这个指标和DMO相辅相成，用于考量不平衡的组织，比如，组织中有两个管理者，但是一个手下有50个人，而另一个只有10个人。指标的值越低，就说明二进制文件的责任更集中，也就是说跨组织协调工作的需求就越少。

- 组织代码责任等级

  这个指标指的是负责某个二进制文件的部门对这个文件做出的修改量占其总修改量的百分比，如果没有专门负责的部门，那么就把修改量最大的部门作为负责部门。这个值越高越好。不同部门往往会在目标、工作方法和文化上有所不同。要进行跨部门的协调工作就必须处理这些分歧，也就更容易出现不成功的构建、同步问题以及代码错误等问题。如果二进制文件有专门的负责部门，那么大部分的修改都应该由这个部门做出。即便没有明确的负责部门，如果大部分的修改都是由某一个部门做出的，那么结果也应该更好。

- 部门总责任

  这个指标是修改某个二进制文件并报告给主要负责人的开发人员数量，和所有修改过这个文件的开发人员数量的比例。我们认为这个值越高，错误就越少。如果大部分做修改的开发人员都是由这个负责人来管理的，那么大部分协调工作的范围就被限定在了这个管理者的旗下，那么协调失灵的情况就会变少。这个指标作为DMO的补充，即较低的DMO（有害）有可能会和较高的部门总责任等级（有利）相互抵消。

- 部门交集度

  这个指标是指对某个二进制文件至少提交了10%的修改的部门数量。这个值越高，说明

二进制文件的部门分布越广，而我们认为这将导致更多的故障。当修改某个二进制文件的部门数量变多时，就难以找出主要负责的部门。而修改这个文件的组织之间，也可能会有相互冲突的目标。而这个文件需要多个部门来修改，也可能表明它是系统的架构中很重要的“连接点”。

这些指标都考量单个二进制文件的组织性复杂度。较高的组织性复杂度意味着组织结构和软件结构很不一致。如果我们根据康威定律做出的推论是正确的，那么当社会和技术结构更“搭调”的时候，就能够产生更好的结果。

那么，在对Windows Vista的发布后故障的研究中，这些指标是否好用？为了回答这个问题，Nagappan、Murphy和Basili使用了一种*逻辑回归*的量化分析法。逻辑回归采用一系列相互独立的变量作为输入，并输出一个分类。在这个案例中，他们计算了每个二进制文件的复杂度指标，作为独立的变量。他们使用了发布后的故障数据来将二进制文件分为容易出错和不容易出错的，然后用逻辑回归来判断这些独立的变量（即复杂度指标）和分类（即是否容易错误）之间有没有关联。此外，使用这个分析法的话，我们还可以*在消除其他复杂度指标的影响的前提下*，找出单个复杂度指标和出错倾向之间的关联性。

这有一点类似于研究某个人的年龄和身高对于他的体重的影响。如果我们分析年龄和体重之间的关联，那么就会发现它们之间有着强烈的联系，身高和体重也是如此。但是，如果我们只研究某个固定身高的人群，就会发现年龄和体重几乎没有什么关系（平均来说，6英尺的美国男人的体重差不多都在210磅左右，无论他是30岁还是60岁）。逻辑回归还可以用来做预测。我们可以对逻辑回归进行“训练”，比如用一群已知年龄、身高和体重的人的数据，来预测某个已知年龄和身高的人的体重。

Nagappan、Murphy和Basili的研究统计结果显示，八种复杂度的指标都对错误有统计上显著的效果，即便是将他们进行横向对比也是一样。也就是说，某个二进制文件的离职开发人员的数量越高，这个文件的错误也就越多，即使不考虑其他的影响因素，如开发人员总数、主负责人深度等。这表明不但复杂度指标和发布后的缺陷有关联，而且每个指标所衡量的东西都有所不同。

下一步，Nagappan、Murphy和Basili尝试着用复杂度指标来*预测*容易出错的二进制文件。做预测比起单纯的找关联来说要困难得多，因为有很多因素都可能影响最终的结果。人们可以确定年龄和重量之间有强烈的关联，但是这并不代表可以在只知道年龄的情况下准确地预测体重。不过，这群研究人员意外地发现，复杂度指标其实可以准确地预测二进制文件的出错倾向。事实上，组织性复杂度指标比起已知的准确性高的源代码特性预测指标（如代码行数、圈复杂度、依赖关系、代码测试覆盖率等）都要好。

要对比不同的预测方法的准确性，可以使用标准的*查准率*和*查全率*分析法。

- 查准率：在这个案例中，查准率代表着使用某个预测方法预测为容易出错的二进制文件有多少确实容易出错。如果这个值低就意味着有很多误报，即被预测为容易出错的二进制文件实际上并不容易出错。
- 查全率：查全率意味着有多少容易出错的二进制文件被预测出来了。如果这个值低，意味着有很多本来很容易出错的二进制文件被错误地预测为不容易出错。

如果可能的话，我们想尽可能地增大查准率和查全率。理想的预测方法应该可以预测出所有真正容易出错的二进制文件，但是又不会错误地包含不容易出错的文件。表11-1为用组织结构回归模型来做预测的查准率和查全率，并有其他源代码相关的指标的预测结果作为对比。

表11-1 预测准确度

预测方法	查准率	查全率
组织结构	86.2%	84.0%
代码变更	78.6%	79.9%
代码复杂度	79.3%	66.0%
依赖关系	74.4%	69.9%
代码覆盖率	83.8%	54.4%
发布前的bug	73.8%	62.9%

给我们的启示

这些结果的意义在哪儿？首先，在Windows Vista的开发中，很明显组织复杂度和发布后的缺陷有着强烈的关联。这些结果证实了我们的推论，即（至少在Windows Vista的开发中）当沟通和协调的结构和软件结构符合的时候，软件就“更好”。当来自不同组织的开发人员开发同一个二进制文件的时候，这个文件就更容易出错。对这种情况我们可以做很多解释。比如，开发人员如果离开公司，他对二进制文件的相关知识就随之带走了，而主要负责人深度越低，就意味着开发同一个二进制文件的开发人员必须通过高层管理者来进行沟通，会带来沟通成本、延迟以及信息缺失等问题。

Nagappan、Murphy和Basili并没有确定究竟哪些因素造成了发布后的缺陷。对于Windows Vista这种级别的项目来说，这种想法太难实现了，而且经济上也不允许。比如说，这可能会需要一个类似于“临床”对比试验一样的环境，一组开发人员在组织结构最简化的环境中开发一个操作系统，而另外一个对照组的开发人员则在更传统、目前常见的组织结构下开发同样的系统。

虽然没有详细的因果分析，但是我们仍然能从这些结果中得到不少收获。如果研究人员认为组织复杂度和发布后缺陷之间有因果关系，那么他们就可以尽量去降低复杂度。比如，与其让组织的所有人都来修改一个二进制文件，不如明确地任命一个负责团队，这个团队可以监管这个二进制文件或者审查提交上来的代码。由于每个软件项目都会存在相互依赖关系的问题，所以我们无法完全避免高组织复杂度的组件，但是可以提前识别出这些组件。在开发的早期就把这些组件的接口稳定下来，有助于减轻组织复杂度在中后期所带来的负面影响。对于可能出错的二进制文件的预测可以用于指导开发后期对测试资源的分布，这样就可以尽量照顾那些更容易在发布后出现错误的组件。

这些研究结果有多可信？要想回答这个问题我们必须了解研究的背景。这个研究对比了具有不同特征的软件组件（二进制文件），但是这些组件都属于同一个软件项目。这样的方法避免了很多干扰因素，因为相对于两个不同项目之间的对比来说，这样的对比更加地对等。试图用分析

不同项目的方式来把某个因素的影响提取出来的做法是不可取的，因为项目与项目之间在很多方面上都不同。结果的差别有可能是所研究的因素造成的，也有可能是因为外部的、没有观察到的因素造成的，比如团队经验、软件的领域或者使用的工具等。在微软的Windows部门，有一个统一的流程，部门上下都是用这套流程和相应的工具，而做决定的方式也大同小异。微软甚至付出了很大的努力来保证不同国家的开发工作也能遵循一个统一而集中的流程。Windows开发环境的统一性降低了对这个研究的内部有效性的质疑，并让我们更确信，Nagappan、Murphy和Basili三人观察到的组织复杂度和发布后缺陷之间的关系并不是由于二进制文件之间的不同所导致的偏差造成的。

而且，这个研究的规模很大，包括上千个开发人员和二进制文件，以及数百万行源代码。由于Brooks观察到的协调和沟通会有超线性增长的情况，我们认为协调失灵的影响在较大型的软件项目中会更大一些。Windows Vista很明显是我们当今使用的软件中屈指可数的大项目，无论是从开发人员数量还是代码的量级。所以说，有理由认为，小一些的（至少在开发人员数量上少一些的）项目，并不会在组织复杂度上受到如此大的影响。这些研究人员们用更小的数据集来重复了他们的实验，以便确定在何种规模下，组织复杂度可以算作好的错误预报机制。他们发现，在30人的开发团队和3层管理层的项目中，仍然可以观察到复杂度指标的效果。

在阅读了研究原文之后我们发现，作者们对于细节非常严谨，并对量化研究中可能出现的问题有深刻的认识（比如研究人员们使用了主成分分析来降低复杂度质变之间的多重共线性对结果的影响），对于研究结果有非常全面的分析。这个假说现在既有了理论支持（康威定律、Brooks定律等）又有了实证证据。此外，在和那些有着软件项目管理经验的人进行讨论之后，他们表示这些结果和他们的直觉及经验相符。由于各种实验和研究都未能反驳我们的结论，作为科学家，我们对这些理论的信心与日俱增。如果能在更多不同背景的项目中重复这个研究是十分有益的，因为这样我们就能知道这些结果是否普遍使用，或是只适用于Windows Vista，还是只适用于某一种类的项目。最后，实践者必须同时考虑这些研究的背景和结果，并对这些研究结果是否符合自己的实际情况做出明智的判断。

11.4　开源软件集市上的小教堂

前两个研究着眼于商业软件。而最近二十年间，一种新型的开发模式逐渐浮出水面。这种我们称之为*自由*或者*开源*的软件开发主要是通过互联网进行，从事开发的各方从未谋面，也通常不存在共同的经济利益。这和“传统”的开发有很多不同。虽然开发人员的雇主会付钱给他们来对这些开源项目进行开发，开源项目本身很少直接付钱给开发人员。也就是说，大多数人可以来来去去，随心所欲。而且由于不存在一个“经营”项目的公司，组织结构也比较松散。

Eric Raymond所写的一系列论文让他成为开源运动的代言人。他认为，这种类型的开发可以被视为是一个临时的*集市*，开发人员们随意地在代码库周围来来去去，随意和他人交谈，并做任何他们愿意的工作。与此相对，他描述了一套可控制而有计划的商业环境下的开发流程，称之为教堂模式，即每个人都被分配到清楚规划好的任务，而所有的任务都已预先安排好。当然，这种

比较极端的非此即彼的分类稍微有些夸张。即便是把开源软件的开发和商业软件的开发视为两种完全不同的方式也不完全准确。

我们已经看到，商业软件的开发常常会遵循康威推论，但是开源软件呢？Brooks定律能够成立的原因之一就是因为大型团队的额外沟通成本。如果开源软件项目真是一个没有结构的集市，那么它又如何来处理随着项目的规模变大而呈超线性增长的沟通渠道问题呢？我们决定对一些著名的开源项目的社会结构进行研究，看看它们是不是真的像一个集市一样，此外，我们还将看看这些项目的组织结构和软件架构之间的关系[1]。

和商业公司不同，开源项目常常没有明确定义的官方组织结构。为了找出这些开源项目的社会结构，我们搜集了开发人员邮件列表的历史档案。这些档案包括了各种讨论，比如各种决议、修正、政策以及所有运营开源项目所必须的沟通。某些开源项目还使用IRC，但是我们所选择的项目要求所有重要的讨论必须通过邮件列表进行。这些讨论的参与者包括项目的开发人员（即可以对源代码储存库进行写操作的人）、提交补丁的贡献者以及其他一些只是想参与讨论功能、bug之类的人。这些项目中大部分的活跃参与者们都使用多个邮件地址，而我们希望能够把这些邮件地址都对应到实际的人身上。为了解决这个问题，我们使用了启发式检测和人工检测来判断电子邮件的地址。此外，我们还把邮件地址对应到了源代码储存库的账号。我们用分析邮件列表的互动并生成社交网络的方式来重构项目的社会组织结构。举例来说，如果张三发表了一条消息到邮件列表，而李四阅读之后发表了回复，那么张三和李四之间就有了信息流动，也就可能会产生合作。因此，我们可以根据这些互动信息来创建社会组织网络。我们用每对参与者之间的互动数量来作为连接他们的边的权重。

我们分析了Perl、Python、Apache服务器、PostgreSQL数据库以及Ant Java编译系统。这些邮件列表的参与者数量从Python的1329到Perl的3621不等，实际的项目开发者数量从Perl的25人到Python的92人不等。表11-2展示的是我们所研究的项目的统计数据。

表11-2　开源项目统计

项　目	Apache	Ant	Python	Perl	PostgreSQL
开始日期	1995-02-27	2000-01-12	1999-04-21	1999-03-01	1998-01-03
结束日期	2005-07-13	2006-08-31	2006-07-27	2007-06-20	2007-03-01
邮件数量	101250	73157	66541	112514	132698
邮件列表参与者	2017	1960	1329	3621	3607
开发人员	57	40	92	25	29

我们使用这些社会网络以及开发人员的修改提交活动数据来回答两个重要的问题。

- 这些参与者是否会按照我们所发现的社会网络自然而然地形成团队？
- 软件的技术结构和参与者的团队结构之间有什么关系？

要回答第一个问题，就需要涉及复杂网络分析的领域。这个领域中的主要研究课题之一是探测*群落结构*，即探测某个网络中存在着的相互紧密连接的子网络。群落结构检测技术尝试把一个网络分成一组一组的节点，这些组内部的节点联系非常密集，而组之间的联系则很稀松。群落结

构的程度则可以被量化为一个指标，即模块度。模块度的范围从0到1，0代表完全随机的网络，1代表网络是由多个互不相关的派系组成。此前已有研究发现，自然产生的具有清晰分组的网络（即模块化的网络）的模块度在0.3到0.7之间。

图11-2的盒状图简单地展示了这些项目在三个月内的模块度。大部分的模块度都远超0.3的阈值，这意味着强烈的群落结构。更具体一点说，虽然这些网络太大太复杂以至于我们无法简单地将其用图表来表示，但是我们可以看出沟通网络是模块化的，有清晰的开发人员分组，组内的开发人员相互之间沟通比起组间的沟通要多得多。这些证据表明开源项目并不是无组织无纪律的开发人员组成的，他们之间也并不是盲目混乱地相互沟通。在开发人员自愿的情况下，似乎（至少对这些项目来说）开源项目实际上也是由有组织、有结构的开发团队完成的。商业软件团队和这些团队的关键区别是开源软件的团队是自然形成的，而不是由管理层决定。此外，我们还发现这些团队的组成要比商业团队更加动态，团队生命期很少超过六个月，并常常以各种不同的方式重组。

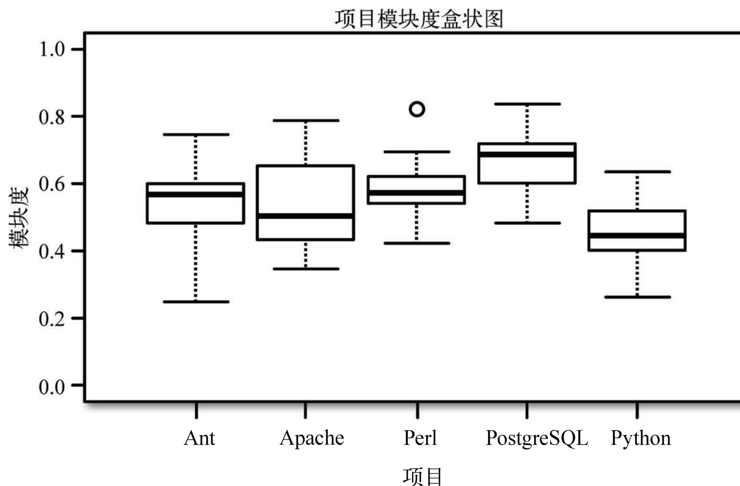


图11-2　模块度（按项目）

仅仅只是证明在沟通网络中存在一个个的分组还不能说我们验证了康威定律或者我们的推论。我们还需要研究这些相互存在沟通的分组之间的关系以及他们共同进行的技术任务。这个社会结构和软件结构是否有联系？要回答这个问题，我们研究了团队的各种开发活动，具体就是每个团队的开发人员所涉及的文件、函数及类。基于对这些信息的分析，我们观察到两点。

首先，相互交流的开发人员所负责的系统部分通常是相互关联的。如果我们的群落检测算法把两个开发人员分到同一个团队，那么他们就非常有可能正在修改同样的一些文件或者函数。此外，在同一个团队的成员之间的邮件沟通中也常常会提到上面的那些函数和文件。也就是说，沟通模式正好反应了开发人员之间的合作开发行为。

其次，我们仔细观察了实际开发工作，以便观察真实情况。本文中的例子较少，如果读者感兴趣可以在完整版论文中找到更深入的分析。

在Apache Web服务器项目中，从2003年的5月到7月，有这样一个团队，组员包括Rowe、Thorpe（这两个是开发人员）以及Deaves、Adkins和Chandran。他们讨论了一些和mod_ssl和Apache的SSL接口相关的bug的修正问题。讨论的话题包括错误的输入/输出代码、模块加载/释放/初始化以及mod_ssl和服务器的整合问题。几乎所有的讨论都是关于SSL的代码，而且在这个时间段之内这个小组内的成员所修改的文件几乎都是在modules/ssl目录之下，如下所示。

modules/arch/win32/mod_isapi.h

modules/ssl/mod_ssl.c

modules/ssl/mod_ssl.h

modules/ssl/ssl_engine_config.c

modules/ssl/ssl_engine_init.c

modules/ssl/ssl_engine_io.c

modules/ssl/ssl_engine_kernel.c

modules/ssl/ssl_engine_pphrase.c

modules/ssl/ssl_toolkit_compat.h

modules/ssl/ssl_util.c

modules/ssl/ssl_util_ssl.c

modules/ssl/ssl_util_ssl.h

support/ab.c

在另一个例子中，2002年10月到12月PostgreSQL项目中的一个团队只进行了关于C语言的嵌入式SQL的工作，另一个团队则完全只关注于更新SGML文档源。在后来的一段时间中，一个新产生的小组的活动和讨论全部关于开发和测试PostgreSQL的JDBC驱动程序（他们修改的源代码和测试代码都处在JDBC的子目录下），而另一个小很多的小组则关注了Unicode的支持。

此外我们还发现了很多类似的例子，这些例子都是项目的参与者们根据某些任务分成小组，而这些任务则是直接对应到代码库的某个部分。关于如何解决问题的讨论、任务的分配、决策的制定的沟通正好可以对应正在进行的实际开发工作，有时候甚至可以直接对应到生成的补丁的结构。显然，对于这些项目来说康威推论是成立的。

那么，这些研究结果的意义在哪？首先，这表明软件结构和社会结构之间的强烈关联和开发流程无关，如果遵循了这些关联，开源的软件项目也能够做得很好。其次，我们了解到开源软件处理扩展性和工作分配问题的方式和传统的开发模式类似。少数开发人员（就我们观察到的来看平均是3到7个人）会组成小组，共同完成软件中的一小部分的相关任务。我们的研究结果支持了一种常见的说法（虽然仍然还有争议），即成功的开源软件项目具备自我调整和自我优化的功能，而这种说法符合康威推论。此外，虽然曾有人认为开源软件项目可以在某种程度上避免Brooks定律和康威定律的影响，但是研究显示这些项目似乎还是会受到两者的影响。

11.5　总结

总的来说，这些研究告诉了我们什么，而我们又应该如何利用这些研究结果来改善现在的软件开发实践呢？

这些研究都证明，当软件项目中的开发人员的组织结构和软件的系统结构类似的时候，项目会比两者不同的时候做得更好。我们在做决策的时候，应该把两种结构都考虑进来，特别是在初期设计和规划的阶段。当经济上允许的时候，我们应该尽量把项目结构和软件结构密切对应起来。甚至在项目结构已经定下来的时候，也应该重新安排结构，已保证二者的密切联系。

康威推论可以为开发带来负面的影响也可以带来正面的影响，而且如果项目负责人能够善用它，就可以把项目做得更好。现在，证据已经被提出，如果还要忽略它的话，就要自己承担风险了。当然，你还需要自己独立的判断。如果软件项目中只有5个开发人员，那么就算违反康威推论的影响估计也不会太大。在软件开发全球化的时代，我们可能很难纯粹为了迁就这个推论而让开发人员搬迁。尽管如此，我们还是相信，在所有背景都相同的情况下，如果我们能更努力和主动地去将软件项目的社会结构和技术结构对应起来，结果仍然会比不去做要好很多。

随着软件团队不断地增大，对于合作和协调的深入了解也将为我们带来越来越大的价值。我们鼓励那些在自己身上观察到了康威推论的正面或者负面影响（或者寻找了影响但是没找到）的软件项目和团队能够分享他们的经验，并让我们对社会和技术结构之间的互动有更多的了解。只有人们都来分享证据或者反证，我们才能继续理解和改善我们的理论。

11.6　参考文献

[1] [Bird et al. 2008] Bird, C., D. Pattison, R. D’Souza, V. Filkov, and P. Devanbu. 2008. Latent Social Structure in Open Source Projects. *SIGSOFT ’08/FSE-16: Proceedings of the 16th ACM SIGSOFT Symposium on Foundations of Software Engineering*: 24-35.

[2] [Brooks 1974] Brooks, F.P. 1974. The Mythical Man-Month. *Datamation* 20(12): 44-52.

[3] [Cataldo et al. 2006] Cataldo, M., P.A. Wagstrom, J.D. Herbsleb, and K.M. Carley. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. *Proceedings of the 20th Conference on Computer Supported Cooperative Work*: 353-362.

[4] [Conway 1968] Conway, M.E. 1968. How do committees invent? *Datamation* 14(4): 28-31.

[5] [Nagappan et al. 2008] Nagappan, N., B. Murphy, and V. Basili. 2008. The influence of organizational structure on software quality: An empirical case study. *Proceedings of the 30th International Conference on Software Engineering*: 521-530.

第12章

测试驱动开发的效果如何

Burak Turhan
Lucas Layman
Madeline Diep
Hakan Erdogmus
Forrest Shull

在软件行业中，测试驱动开发（TDD）[1]是最常被谈到，却又最少被用到的敏捷实践之一。它被忽视的主要原因是我们不理解它对人、流程、产品的作用。虽然大部分人认同在编码之前写测试能促进更稳健的代码实现和更好的设计，与TDD效果相关的未知成本，以及对程序员最常用的“先编码再测试”模式的倒置，阻止了TDD的广泛使用。

为了提供当前TDD效果的证据总览，我们开展了一次对在线数据库和科学出版物中TDD研究的系统性评审。系统性评审是一种在医疗领域中非常普及的研究方法，用来汇集并分析临床试验的数据结果。系统性评审旨在回答普遍性问题：“发表的证据如何证明使用技术X的作用？”在医学中，系统性评审在评估药品效果和对疾病的替代疗法中扮演着关键的角色。实证软件工程研究员们也采用了这种方法来总结和分析各种软件开发实践的效果的证据。这一话题已经在第三章中有所描述，Barbara Kitchenham著。另外，在她[4]和Dybå等人[3]的报告中也有所谈及。

在这章中，我们不会提供正式的系统性评审报告，而会把TDD想象成一种药丸，从药理性的角度来描述它的疗效。我们希望读者把这章剩余的章节想象成TDD“药丸”的医疗报告，并带着以下问题继续阅读：

“如果TDD是一颗药丸，你会服用它来提高你的健康指数吗？”

12.1 TDD 药丸是什么

以下列出的是TDD药丸的配方，必须“严格”按照顺序一步一步执行：

(1) 选择一项小任务；

(2) 为这项任务写一个测试；

(3) 运行所有测试来验证新的测试失败；

(4) 写出最少的生产代码来完成任务；

(5) 运行所有测试（包括新的测试）来验证它们通过；

(6) 按照需要重构代码；

(7) 从第(1)步开始重复。

TDD药丸中起主要作用的成分是在产生代码前创造测试用例。在写代码前创造测试用例需要有足够的耐心来考虑设计解决方案：信息的流向、代码可能的产出、以及可能发生的异常情况。在产生代码之前运行新写的测试用例能帮助验证新写的测试用例是否正确（如果测试用例此时已经通过，那就意味着没达到期望的测试效果），而且系统是可以正常编译的。TDD药丸也包括只编写恰到好处刚好能使测试用例通过的生产代码，以便促进整洁的模块化设计。另外，TDD使用者创造了一个不断增长的自动化测试用例库，一旦现有系统发生了改动，可以随时运行所有的测试库来验证改动的正确性。

正如其他许多药物一样，TDD药丸也有一些官方的变种，包括ATDD（验收测试驱动开发）、BDD（行为驱动开发）和STDD（故事测试驱动开发）。ATDD把一步步的“小任务”替代为“功能层面的业务逻辑任务”，而BDD则使用“行为规范”。TDD中任务的顺序使之区别于其他治疗方法，但是TDD药丸的官方变种也会包含一些其他配方，如将任务拆繁为简、重构、保持测试-编码的短循环、以及持续不断的回归测试。

警告 除了重构之外，由于实践中的不同解读和主要成分的支配性，一些关键配方可能不存在于许多“通用的”TDD药丸中。

12.2 TDD 临床试验概要

我们回顾的重点是收集TDD药丸对代码内部质量（参照下面的“度量代码质量”）、外部质量、生产力和测试质量所产生效果的定量的证据。对TDD药丸的评估基于从32项临床试验中收集的数据。在2009年的第一季度，作者从在线综合指数、主流科技出版物（ACM、IEEE、Elsevier）和“灰色文献”（技术报告、论文）中收集了325项TDD研究的报告。然后，这325项报告通过两级筛选程序被缩减为22项。4位研究者过滤掉了2000年之前开展的研究、TDD药丸的定性研究，以及问卷调查和完全主观的分析。这些报告中有些包含多项或重叠的试验（也就是说，同样的试验出现在不同的文章中）。在这种情况下，试验仅被计算一次。之后，由5位研究者组成的团队从报告中抽取了关于研究设计、研究背景、参与者、治疗方法及效果对照，以及研究结果的关键信息。研究团队总共分析了22篇报告，32项独立试验。

度量代码质量

系统的“内部质量”与它的设计质量有关。通常的理解是，好的设计意味着简单、模块化、易于维护和易于理解。虽然TDD主要被理解为开发实践，它也被认为是设计实

践。当使用TDD药丸时，我们期待出现增量式的简单设计。简单设计是由下列实践驱动的：第一，模块化，它使代码可测；第二，写最少的生产代码来完成简单的任务；第三，持续重构。TDD试验用以下的一种或多种方法来评估系统的内部质量：

- 面向对象的指标。这涉及每类加权方法数（WMC①）、继承树的深度（DIT②），等等；[2]
- 圈复杂度；
- 代码密度（如，每个方法的代码行数）；
- 每个功能的代码量。

系统的“外部质量”通常由发布前或者发布后的缺陷数量来度量。TDD被认为可以提高外部质量，因为它鼓励开发人员多写测试用例，这点对于简单任务的开发人员很容易理解，系统被不断地回归测试，而由改变引起的错误能很容易地被细粒度的测试检测到。在TDD的试验中，我们用以下一种或多种指标来汇报外部质量：

- 通过的测试用例数；
- 缺陷数目；
- 缺陷密度；
- 每个测试的缺陷数；
- 修改缺陷的工作量；
- 变更的密度；
- 预防性变更的比例。

32项试验以对比实验、试点研究或商业项目的方式在学术界或产业界中展开。对比实验在学术实验室或者可控产业环境中通过定义好的研究协议展开；试点研究使用不那么结构化的实验任务执行；商业项目启用把TDD作为每日工作一部分的行业团队。这些试验的参与者有不同的经验等级，从大学生到研究生到专业人员。每项试验参与者的人数从1个人到132个人不等。花在试验上的工作量也有很大的跨度，从几个人小时到21 600人小时。每个试验都比较了TDD药丸与其他治疗方法（通常是传统的最后测试的开发方法）相比的效果。参与群体的对象也由不同单元组成，如个人、结对、团队和项目。

基于参与者的经验、实验结构的细节、试验的范围，我们把32个试验分为4个级别。参与者的经验由他们是否是本科生、研究生或专业人员决定。

基于对TDD的使用过程和对比疗法的描述，我们将试验的构建评估为好，一般，差或者未知。“好”的构建执行所有TDD处方中的成分，“一般”的构建规定了先写测试但未包含所有的TDD成分，“差”的构建没有执行TDD的步骤，“未知”的构建没有说明TDD步骤是否执行。

最后，基于参与时间和参与人数所得出的汇报工作量或预估的工作量，我们把试验的规模记录为小、中、大。小项目涉及少于170人小时的总工作量，而大项目的范围从3000人小时到21600

---

① Weighted Methods Per Class：每个类所定义的方法的个数，度量一个类中所定义方法的复杂性。——译者注

② Depth of Inheritance Tree：对根类的最大继承路径。——译者注

人小时不等。我们使用了一个简单的类聚算法把项目的规模分类，而参与者的经验和构建的细节则基于试点报告中发现的描述性数据。

随着试验级别的提升，我们也对使用TDD药丸在“现实生活”中的效果更有信心。最低的级别是L0，只包含所有的小规模试验。这些试验只汇报了小于170人小时的工作量或少于11个参与者的情况。下一个级别是L1，包含中等或大规模的试验，但是实施的构建被评为“差”或“未知”。L2级别包含了被评为“一般”或者“好”，并且参与者为本科生的大中型试验项目。最高的级别为L3，包含被评为“一般”或者“好”，并且参与者为研究生或专业人员的大中型试验项目。

表12-1总结了我们对试验分级时所使用的参数。表12-2展示了每个级别中试验的个数。

表12-1 临床TDD试验的等级

	L0	L1	L2	L3
经验	任意	任意	本科生	研究生或专业人员
构建	任意	差或未知	还行或好	还行或好
规模	小	中或大	中或大	中或大

表12-2 临床TDD试验的分类

类　别	L0	L1	L2	L3	总　数
对比试验	2	0	2	4	8
试点研究	2	0	5	7	14
行业应用	1	7	0	2	10
总数	5	7	7	13	32

12.3 TDD 的效力

我们分析了TDD试验中TDD药丸对于生产力、内部和外部质量、测试质量效果的定量结果。直接比较跨试验的定量结果是不可能的，因为不同的试验用不同的方法度量TDD的效力。所以，我们对每个试验分配了一个概要值：“更好”、“更差”、“混合”或者“无法确定/没有区别”。概要值由TDD药丸疗效的定量结果与传统对照组的比较所决定。概要值也包含了报告的作者对试验结果的解释。在概要值为“更好”的试验中，大多数量化指标证实TDD药丸比对照疗法的疗效更好。在概要值为“更差”的试验中，大多数量化指标支持对照疗法的疗效更好。概要值为“无法确定/没有区别”的试验没有决定性的结果，或者没有观察到区别。最后，概要值为“混合”的试验中，一些度量支持TDD，而另一些不支持。所有这些概要值都参考了研究者对于研究结果的解读，因为在很多情况下，报告省略了一些可能会影响客观外部评估的细节。

在以下章节中，我们会尽力得出一些关于试验中TDD价值的结论。

12.3.1 内部质量

试验中得到的证据显示，TDD对内部质量效果并不一致。虽然TDD在一些指标类型（复杂度

和重用度）中似乎相比对照实验组产出了更好的结果，但是在其他指标（耦合和内聚）上TDD的表现总是差强人意。另一个从试验数据中观察到的结果表明，TDD产出的生产代码在方法/类的级别更简单，但是在包/项目级别更复杂。这种不一致的效果在更严格（即L2和L3）的试验中更明显。内部质量的区别也可能是由于其他因素造成的，如积极性、技能、经验和学习效果。表12-3基于内部质量的度量对试验归类。

注意 在下列表格中，每一格中的第一个数字汇报了所有试验，而括号中的数字仅汇报L2和L3的试验。

表12-3 对于内部质量的效果

类 别	更 好	更 差	混 合	无结果或无区别	总 数
对比试验	1(0)	0(0)	0(0)	2(2)	3(2)
试点研究	1(1)	1(1)	3(1)	2(2)	7(5)
行业应用	3(1)	1(1)	0(0)	0(0)	4(2)
总数	5(2)	2(2)	3(1)	4(4)	14(9)

12.3.2 外部质量

有一些证据显示TDD能改善外部质量。虽然对比试验的结果几乎都是非结论性的，但行业应用和试点研究都强烈偏向于TDD。然而，行业应用和对比实验对TDD的偏好性证据在过滤掉不严格的研究（L0和L1试验）后消失了。另外，试点研究和对比试验的证据在L0和L1被过滤掉之后显示出矛盾。然而如果所有研究都同等计数的话，证据会显示TDD药丸能改善外部质量。表12-4基于外部质量度量对试验归类。

表12-4 对于外部质量的效果

类 别	更 好	更 差	混 合	无结果或无区别	总 数
对比试验	1(0)	2(2)	0(0)	3(3)	6(5)
试点研究	6(5)	1(1)	0(0)	2(2)	9(8)
行业应用	6(0)	0(0)	0(0)	1(1)	7(1)
总数	13(5)	3(3)	0(0)	6(6)	22(14)

12.3.3 生产力

“生产力”引发了关于TDD最有争议话题。虽然许多人承认使用TDD需要一条陡峭的学习曲线，可能会在一开始降低生产力，然而人们并没有对长期效果的共识。一些论点预计生产力会随着TDD的使用而上升，原因包括更容易在简单的任务间切换背景，提高的外部质量（更少的错误以及错误可以更快被发现），提高的内部质量（由于简单设计修改代码更容易），以及提高的测试质量（由于自动化测试，引入新错误的机会很小）。另外一些论点争论说TDD招致太多的额外开

销，会对生产力产生负面效应，因为太多的时间和精力被花在写测试而不是增加新功能上。在TDD试验中所使用的评估生产力的不同度量方法包括：开发和维护成本、随时间产出的代码量或者功能量、每单元工作量所产出的代码量或功能量。

试验中得到的证据显示，TDD对生产力的效果并不一致。对比试验中的证据显示出生产力随着TDD的使用而提升。然而试点研究提供了混合证据，一些支持TDD，另一些反对。在行业研究中，证据显示TDD使生产力更差。即使只考虑更严格的研究（L2和L3），证据也被分为支持和反对在生产力上的效果。表12-5基于生产力的效果对试验归类。

表12-5　对于生产力的效果

类　别	更　好	更　差	混　合	无结果或无区别	总　数
对比试验	3(1)	0(0)	0(0)	1(1)	4(2)
试点研究	6(5)	4(4)	0(0)	4(3)	14(12)
行业应用	1(0)	5(1)	0(0)	1(0)	7(1)
总数	10(6)	9(5)	0(0)	6(4)	25(15)

12.3.4　测试质量

由于在TDD中测试用例先于所有开发行为，测试一个不断发展的系统的正确性应该在成长中的自动化测试的帮助下更容易。此外，由于细粒度测试的产出，测试过程应该是高质量的。在试验中，测试质量由测试密度、测试覆盖率、测试生产力或测试工作量所决定。

有一些证据显示TDD改善了测试质量。大部分证据从试点研究中来，即使在过滤掉不严格的研究之后也显示出支持TDD。对比实验显示TDD至少和对照疗法有相同的进展。行业应用中的证据不足，无法做出结论。

因此，TDD相关的测试质量似乎至少不比其他方法差，而且通常要更好。这里我们期望更有力的结果：既然鼓励测试用例开发是TDD起作用的成分中最主要的一个，总体的证据对TDD的支持应该在这些研究报告的测试质量度量中显现出来。

表12-6基于测试质量对试验归类。

表12-6　对于测试质量的效果

类　别	更　好	更　差	混　合	无结果或无区别	总　数
对比试验	2(1)	0(0)	0(0)	3(3)	5(4)
试点研究	7(5)	1(1)	0(0)	1(1)	9(7)
行业应用	1(0)	1(1)	0(0)	1(0)	3(1)
总数	10(6)	2(2)	0(0)	5(4)	17(12)

12.4　在试验中强制 TDD 的正确剂量

虽然大部分试验并没有控制TDD药丸的服用量（在软件说法中被翻译为缺乏对过程一致性的

注意），我们相信不同的试验和不同的对象所使用的剂量是不同的。“差”和“未知”构建的试验可能没有严格执行TDD的使用方法，我们也相信很可能试验参与者选择一些成分定制了药丸而不是严格遵循TDD的教科书定义。这个问题对得出普遍结论产生了威胁。在医学环境中，不强制执行或度量TDD的使用类似于没有保证病人在治疗中服药，或者不知道病人服用的剂量。因而，所观察到的TDD药丸的效果可能归咎于流程的不一致性或者其他一些没有被足够描述和控制的因素。在将来的试验中，治疗方法的一致性和对照组应该受到仔细的监控。

不管TDD试验的汇报质量如何，有人会提出一个相关的问题：“在现实生活中需要遵循TDD的书本定义吗？”有时候在为特定工作背景和个人风格做出调整后，病人服用半颗或四分之一颗药丸就会有起色。有微观层面的开发日志工具可用于调查这些问题。这些日志工具可以帮助控制TDD流程的一致性，并帮助理解TDD在现实生活中的实践执行方式。

12.5　警告和副作用

在这一节中，我们将提出几个关于TDD药丸的问题，可能可以调和TDD在不同环境下的效力。

- 会对环境起反应吗

  对TDD的使用没有推荐的最佳环境。我们不知道它是否对所有领域、一个领域内的所有任务、各种大小和复杂度的项目均适用。比如，试验没有澄清TDD实践是否对开发嵌入式系统或高分散系统适用，在这些系统中增量式测试可能不切实际。此外，TDD在遗留系统中的使用一直是个问题，这样的系统可能需要对现有代码大量的重构之后才会变得可测。

- 对每个人都适用吗

  一个人人都认同的基本事实是：学习TDD很困难。它伴随着一条很陡峭的学习曲线，需要技巧、成熟度和时间，特别是当开发人员已经形成了编码再测试的习惯定式之后。如果有更好的产生测试用例的工具，或在教育过程中更早地传播测试再编码的习惯，也许可以鼓励TDD的使用。

- 会上瘾吗

  我在与TDD开发者的私下交流中得知，它是一个上瘾的实践。它改变了人们的思维习惯，也改变了人们的编码方法，很难回退。因此，放弃TDD实践与采用它们一样困难。

- 会与其他药物互相影响吗

  我们选择的研究中没有特别针对TDD与别的药物同时服用时的效果的。一个试验显示，在与预先设计结合使用时，TDD提升了40%的外部质量[24]。另一个试验对比了单人和结对开发人员，他们实践了TDD和最后测试的增量式开发方法[15]，那个试验结果显示在软件外部质量上没有区别。我们还不知道哪个实践与TDD合用会产生好或坏的效果。虽然也许有些实践促进了那些预期效果，也有另外一些抑制了它们。之前提到的例子可能是基于具体案例的，但是它们指出了需要对TDD与其他药物交互作用的进一步调查。

12.6　结论

TDD的效果仍然牵涉许多未知数。确实，对于我们所使用的任何度量TDD效果的方法，包括内部质量、外部质量、生产力和测试质量，得出的证据都不是完全一致的。多数的矛盾可能来自于TDD试验对内在因素的不完全描述。因此，TDD仍然注定是辩论和研究的争议性话题。

对寻求一些实践建议的从业者来说，我们的专家小组推荐使用TDD药丸，小心监控它的反应和副作用，并相应增加或减少剂量。所以，我们在总结数据后由小组中的个人提出了一些处方。

> 我们虽然已经汇集了所有证据，但是每个读者必须得出自己的想法。首先，判定哪个特性对你来说最重要。比如，你是否更关注生产力或外部质量？你能证明花更多精力创造更高质量的测试是值得的吗？这章提出的证据只能用于读者基于自己的目标做出自己的决定。
>
> 我已经使用了TDD药丸，而且已经上瘾了。我的个人经验是，TDD能增加生产力，虽然这份研究缺乏这方面的证据。也许我的想法只是一种感觉。基于这些结果，特别是关于对外部质量正面影响的证据，如果我没有准备好使用TDD，我会让我的团队先服用小剂量，然后看看他们自己是否感觉会有长期生产力的增长。如果没有不良反应，我会慢慢增加药量并持续观察。
>
> 虽然TDD很有前景，对它效力的不确定性和前期使用的高成本会阻止它的使用。尽管如此，它的成分似乎能鼓励好的编码和编程习惯的产生，长期来说能产生更高质量的程序员和测试。
>
> 虽然看起来很有前途，但是有些事实我们必须面对，它初尝时很苦。很多人更喜欢旧东西。毕竟，当你花大把时间写失败测试用例的时候很难感到有生产力。从另一个角度说，我这辈子从未写出过更干净的代码了。当我改变旧代码，点击“运行测试”按钮，并确信我没有破坏任何东西的时候，感觉棒极了！
>
> 这章中包含的证据只能证明TDD可能治好你，但是你不应该把它当成万能药。你的TDD征途可能因为某些因素而变化，包括你的经验和你的工作环境。作为一个从业者，何时应该期待TDD带来改进是一种应该培养的洞察力，它也会是一份很有价值的资产。

12.7　致谢

Janice Singer博士是参与系统性评审初期阶段筛选研究的研究者之一。我们非常衷心地感谢她对此次工作做出的贡献。

12.8　参考文献

[1] [Beck 2002] Beck, Kent. 2002. *Test-Driven Development: By Example*. Boston: Addison-Wesley.

[2] [Chidamber et al. 1994] Chidamber, S.R., and C.F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20(6): 476-493.

[3] [Dybå et al. 2005] Dybå, Tore, Barbara Kitchenham, and Magne Jørgensen. 2005. Evidence-Based Software Engineering for Practitioners. *IEEE Software*22(1): 58-65.

[4] [Kitchenham 2004] Kitchenham, Barbara. 2004. Procedures for Performing Systematic Reviews. Keele University Technical Report TR/SE0401.

[5] [Canfora et al. 2006] Canfora, Gerardo, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio. 2006. Evaluating advantages of test-driven development: A controlled experiment with professionals. *Proceedings of the ACM/IEEE international symposium on empirical software engineering*: 364-371.

[6] [Erdogmus et al. 2005] Erdogmus, Hakan, Maurizio Morisio, and Marco Torchiano. 2005. On the Effectiveness of the Test-First Approach to Programming. *IEEE Transactions on Software Engineering* 31(3): 226-237.

[7] [Flohr et al. 2006] Flohr, Thomas, and Thorsten Schneider. 2006. Lessons Learned from an XP Experiment with Students: Test-First Needs More Teachings. In *Product-Focused Software Process Improvement: 7th International Conference, PROFES 2006, Proceedings*, ed. J. Münch and M. Vierimaa, 305-318. Berlin: Springer-Verlag.

[8] [George 2002] George, Boby. 2002. *Analysis and Quantification of Test-Driven Development Approach*. MS thesis, North Carolina State University.

[9] [Geras 2004] Geras, Adam. 2004. *The effectiveness of test-driven development*. MSc thesis, University of Calgary.

[10] [Geras et al. 2004] Geras, A., M. Smith, and J. Miller. 2004. A Prototype Empirical Evaluation of Test-Driven Development. *Proceedings of the 10th International Symposium on Software Metrics*: 405-416.

[11] [Gupta et al. 2007] Gupta, Atul, and Pankaj Jaloye. 2007. An Experimental Evaluation of the Effectiveness and Efficiency of the Test-Driven Development. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*: 285-294.

[12] [Huang et al. 2009] Huang, Liang, and Mike Holcombe. 2009. Empirical investigation towards the effectiveness of Test First programming. *Information & Software Technology* 51(1): 182-194.

[13] [Janzen 2006] Janzen, David Scott. 2006. *An Empirical Evaluation of the Impact of Test-Driven Development on Software Quality*. PhD thesis, University of Kansas.

[14] [Kaufmann et al. 2003] Kaufmann, Reid, and David Janzen. 2003. Implications of test-driven development: A pilot study. *Companion of the 18th annual ACM SIGPLAN conference on objectoriented programming, systems, languages, and applications*: 298-299.

[15] [Madeyski 2005] Madeyski, Lech. 2005. Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. *Proceedings of the 2005 Conference on Software Engineering: Evolution and Emerging Technologies*: 113-123.

[16] [Madeyski 2006] Madeyski, Lech. 2006. The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design—An Experiment. In *Product-Focused Software Process Improvement: 7th International Conference, PROFES 2006, Proceedings*, ed. J. Münch and M. Vierimaa, 278-289. Berlin: Springer-Verlag.

[17] [Madeyski et al. 2007] Madeyski, Lech, and Lukasz Szala. 2007. The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study. *Software Process Improvement, 4th European Conference, EuroSPI 2007, Proceedings*, ed. P. Abrahamsson, N. Baddoo, T. Margaria, and R. Massnarz, 200-211. Berlin: Springer-Verlag.

[18] [Muller et al. 2002] Muller, M.M., and O. Hagner. 2002. Experiment about test-first programming. *Software, IEEE Proceedings* 149(5): 131-136.

[19] [Nagappan et al. 2008] Nagappan, Nachiappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. 2008. Realizing quality improvement through test-driven development: results and experiences of four industrial teams. *Empirical Software Engineering* 13(3): 289-302.

[20] [Pancur et al. 2003] Pancur, M., M. Ciglaric, M. Trampus, and T. Vidmar. 2003. Towardse mpirical evaluation of test-driven development in a university environment. *The IEEE Region 8 EUROCON Computer as a Tool* (2): 83-86.

[21] [Siniaalto et al. 2008] Siniaalto, Maria, and Pekka Abrahamsson. 2008. Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study. In *Balancing Agility and Formalism in Software Engineering*, ed. B. Meyer, J. Nawrocki, and B. Walter, 143-156. Berlin: Springer-Verlag.

[22] [Slyngstad et al. 2008] Slyngstad, Odd Petter N., Jingyue Li, Reidar Conradi, Harald Ronneberg, Einar Landre, and Harald Wesenberg. 2008. The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components—An Industrial Case Study. *Proceedings of the Third International Conference on Software Engineering Advances*: 214-223.

[23] [Vu et al. 2009] Vu, John, Niklas Frojd, Clay Shenkel-Therolf, and David Janzen. 2009. Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project. *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*: 229-234.

[24] [Williams et al. 2003] Williams, Laurie, E. Michael Maximilien, and Mladen Vouk. 2003. Test-Driven Development As a Defect-Reduction Practice. *Proceedings of the 14th International Symposium on Software Reliability Engineering*: 34.

[25] [Yenduri et al. 2006] Yenduri, Sumanth, and Louise A. Perkins. 2006. Impact of Using Test-Driven Development: A Case Study. *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006*: 126-129.

[26] [Zhang et al. 2006] Zhang, Lei, Shunsuke Akifuji, Katsumi Kawai, and Tsuyoshi Morioka. 2006. Comparison Between Test-Driven Development and Waterfall Development in a Small-Scale Project. *Extreme Programming and Agile Processes in Software Engineering, 7th International Conference, XP 2006, Proceedings*, ed. P. Abrahamsson, M. Marchesi, and G. Succi, 211-212. Berlin: Springer-Verlag.

第13章

为何计算机科学领域的女性不多

Michele A. Whitecraft
Wendy M. Williams

本身就会出错的思维框架不能用来解决问题。

——爱因斯坦

这里有一组统计数字：女生从幼儿园到大学的成绩都比男生高，包括数学。根据我们所能掌握的最新的数据来看，大学的数学专业学生中，女性占了48%，参加先修课程（AP）考试的考生中女性占了56%，参加微积分AP考试的考生中，女性也占到了51%[12]。然而，同年计算机科学AP考试的考生中，女性却只占到了17%[12]。

同样的，虽然2008年所有本科学位获得者中女性占了57%，但是计算机科学（CS）和信息技术（IT）的学位获得者中，女性只占18%[33]。奇怪的是，23年前（1985年）计算机科学学士学位获得者中，女性的比例曾高达37%[33]。从2001年至2008年，计算机科学本科的女性入学人数下降了79% [21]。

为什么计算机科学中女性这么少呢？我们应该关心这个现象吗？如果答案是肯定的话，我们可以做些什么来扭转这种趋势呢？对这些问题的争论分为三大类。

一些人认为，女性相较于男性而言，不太可能拥有极端优秀的认知能力，而这种能力在计算机科学领域很重要[8] [9] [18]。

也有人说，女性对计算机科学不感兴趣，喜欢研究其他领域[16] [14] [37]。还有人认为，是各种成见和偏见以及“男性文化”把女性赶出了这个领域[33] [3]。

本章回顾了与这三个观点相关的研究，并探讨其意义。

13.1　为什么女性很少

首先让我们研究一下这个问题的通常解释以及相关的研究。

13.1.1　能力缺陷，个人喜好以及文化偏见

对于在科学、技术、工程和数学领域（STEM）阳盛阴衰的问题，人们已经做了大量的研究，包括先天能力差异、个人喜好和文化偏见等原因。Ceci、Williams和Barnett开发了一种框架，研究各种因素之间的相互影响[10]。接下来，我们将研究每个因素，然后用Ceci等人的整合框架来综合分析。研究得出的图表（见图13-1）让人们感到这些因素之间存在非常复杂的相互影响。虽然生物学意义上的性别差异会起一定的作用，但研究表明，也有可能与不良的性别歧视有关，这就提出了进一步的问题。

1. 女性数学空间能力缺陷的证据

研究人员已经对男性和女性先天能力的差异（以及孩童时代的经历及环境导致的差异）进行了探索，并把它作为一种可能的原因来解释为什么女性在计算机相关领域越来越少。有大量证据表明女性在数学密集型任务上不如男性。这种性别不对称存在于能力分布的最上端。例如，SAT数学成绩前1%的学生中男女比例是2比1，前0.01%学生中的比率是4比1[23] [28]。但分数最低的学生中，男性也占大多数，这意味着男性整体表现差异很大。

Ceci、Williams和Barnett[10]把认知性别差异的证据分为平均差异（分布的中点），右尾差异（最优秀的10%，5%和1%），后者更好地代表了从事科学、技术、工程和数学（STEM）的人群。根据美国1960年~1992年对青少年的概率抽样，Hedges和Nowell发现男性和女性考生的考试成绩分布在最高和最低的1%，5%和10%中差异很大[20]。男性在科学、数学、空间推理、社会研究和机械技能上表现出色。女性在口头表达能力、联想记忆表现和感知速度上胜出。这些发现提出了一个可能性，CS和IT相关领域的性别分布可能和生理特性有关。

大脑的相对大小、脑组织、荷尔蒙差异的研究也与之相关。Ceci和Williams查看了近期生物学上有关认知性别差异的研究，调查了大脑体积、脑组织和激素的差异[8]。Deary等人发现了智力与脑容积的适度相关性（0.33~0. 37）[13]，而男性平均大脑容量大于女性。在讨论这项研究时，Ceci和Williams指出：“大多数关于性别差异的生物研究关注的是平均值，而STEM领域的性别差异研究关注的是最右尾部值（最高的1%，甚至最高0.1%或最高0.01%）。”换句话说，大脑平均差异的研究并不切合我们的问题，因为能证明男女之间数学和空间能力差异的有力证据，只会出现在能力分数范围的最顶部（或底部）。

其他被Ceci和Williams引用的研究，指出男性和女性使用不同的大脑部位来完成相同的任务[17]。Ceci和Williams得出结论：“基于其他独立的重复实验和有代表性的取样，可以总结得出男性和女性使用不同的大脑结构来实现相同的一般认知能力。”

此外，Ceci和Williams引用了一项研究，该研究考察了出生前和出生后激素对于认知性别差异的影响。在这项研究中，雄性老鼠比雌性老鼠更快走出迷宫。一旦雄性老鼠被阉割，他们的优势就消失了。Ceci和Williams还回顾了另一个研究，在变性手术时，女性如果服用大量的抑制雌激素药物，并摄入大剂量的男性荷尔蒙，会增强空间能力。这一领域的大量研究指出，荷尔蒙可能会影响妇女的专业选择。不过，目前还不清楚影响会有多大。Ceci和Williams认为：“没有足够强有力和一致的证据能表明激素是STEM领域性别差异的主要原因。”

在结束荷尔蒙差异的讨论前，我们应该考虑它是否导致了一些行为差异，使得计算机相关工作对女性的吸引力没有对男性大。

统计表明，妇女都致力于专业工作。2008年，美国57%的专业工作由女性承担[4][33]，同时她们在数学这门与计算机密切相关的学术学科中也十分成功（按成绩来衡量）。由此看来，我们似乎需要跳过能力不足的解释，来询问女性自己的选择。统计结果表明，我们需要对影响女性决定是否从事计算机领域的因素进行性别敏感研究。我们还需要考虑女性是否认为自己在CS的男性文化下被剥夺了权利。如果，这里真的藏有性别失衡的重要原因，那么这里也可能存在一个机会来扭转这一趋势。

2. 个人喜好和生活方式选择所起的作用

与此同时，一些研究人员强调了个人喜好和文化的影响。有人声称，人们对职业和生活方式选择的固有印象是女性较少选择计算机科学的主要原因，更有人强调，反对女性选择这类职业的文化压力是最大的原因。接下来，我们将检验这些观点的证据。

关于职业选择，职业上的性别交替在历史上一直出现，尤其是在教学，文秘，医药领域上[8]。这些交替很容易解释，因为随着时间的推移，这些职业的威望和报酬发生了变化，而不是受激素或基因影响。男性不断接手被认为更具有经济价值的工作，这表明性别劳动力模式更多地受到文化和政治力量的驱动，而不是简单的生物差异。在最近有关女性选择从事健康相关职业的纵向研究中，我们发现了一个文化价值驱动职业选择的有趣平行案例。Jacqueline Eccles和美国密歇根大学的同事们发现，即使考虑到数学能力，年轻女性更喜欢从事健康相关的职业，因为较之男性，她们认为面向人和社会的职业更有意义[15]。

Margolis、Fisher 和Miller [30]在他们于2000年进行的研究中，更为深入地证明了女性更倾向于（或更重视）服务他人和社会的选择。该研究涉及卡耐基梅隆大学计算机专业的51个男生与46个女生（总共210个访谈）。这里引述受访者中一位女性的话，呼应了Eccles的理论。

> 我的想法是，你可以拯救生命，而不是从社会中脱离出去。这实际上是把自己作为社会的一份子。这实际上是帮助别人。因为我内心深处想要提供帮助。我觉得我在计算机科学中的唯一的问题是，我将会脱离于社会，而无法对社会做出什么贡献；也无法帮助第三世界国家的人民……我想找到一种方式可以帮助别人，那样我就愿意以计算机科学为职业了。

Margolis、Fisher和Miller发现，女性在计算机中追寻以人为本的目标，这与计算机科学领域的其他研究相一致[22][31][36]。他们的调查指出，44%的女学生（相对于9%男性学生）强调用更人性化的项目整合人与计算机的重要性。总体来看，女性更倾向于医疗用途（如心脏起搏器，肾透析机，以及找出疾病）、通信、解决社会问题的计算，而不是单纯为了计算而计算、为了开发更好的计算机或编写游戏而计算。

Ferriman、Lubinski和Benbow指出了一些相似的价值观问题，生活方式偏好和生活目标取向中的性别差异，是女性在STEM领域中人数较少的主要原因[16]。他们的研究别具一格，因为他们能够把“能力”保持为常量，并把人数范围缩小到STEM领域中最出色的人群。通过对数学早熟

青年超过20年的跟踪，他们发现，“继完成研究生学位后，男性更关注职业、更个体性，而女性似乎在生活目标取向中更全面和公共，更普遍地注意家庭、朋友、自己和他人的社会福利。” 那么按这种说法，CS中女性很少是因为她们对其他学科和领域更感兴趣。

13.1.2　偏见、成见和男性计算机科学文化

很多研究人员拒绝接受女性内在特质（无论能力或兴趣）导致了CS和IT界女性人数很少这一说法。他们认为，真正的原因是CS的文化不鼓励女性加入。在“计算机科学本科女生兴趣剖析”一文中，Margolis、Fisher和Miller关注那些来读计算机时兴趣高涨的女生，如何很快在这门学科中丧失了能力和兴趣。他们探讨了智力之外的因素是如何在抽象的知识体中影响兴趣的。例如，他们探讨了带有性别偏见的标准是如何侵蚀信心的，以及一个男性化的成功标准是如何影响女性的兴趣和能力的。作者认为，可能会有一些“有害方式，其中男性行为方式和兴趣成为‘正确合适’和成功标准”，这反过来导致了女性对这门学科热情的减弱。换句话说，正如他们的访谈所显示，女性如果拒绝使自己符合“为了入侵而入侵的黑客”这一“电脑怪杰”形象的话，会被认为是脱离这个群体的。

对那些认为计算机科学文化就是“神童”们一个个夜以继日独自狂热编程的人，Margolis、Fisher和Miller引用了女计算机科学老师的一段话：

> 我的观点是，熬夜做事情是表示专心以及对这个学科的热爱，也可能意味着不成熟。女孩们对电脑和计算机科学的热爱表现得非常不同。如果你正在寻找这种着魔的行为类型，你是正在寻找一个典型的青年男性的行为。虽然有些女生也会如此，但大部分不会。不过，这并不意味着她们不爱计算机科学！

Margolis、Fisher和Miller案例的缺陷在于他们只研究了修读计算机科学的一小部分人。因此，对于把他们的结论推广到更广泛的群体，我们持谨慎态度。我们不能基于小样本群得出广泛适用的论断。此外，尽管他们的面试问题是为了使学生根据自己的经验而非抽象思维回答，作者们自己也承认了这种采访技巧不利于对不同的独立因素分配相对权重，因为“因素经常转移而且似乎相互纠缠”[30]。

同时，这些研究结果与其他对于计算机文化的研究相呼应，比如一个由美国大学妇女协会（AAUW）的教育基金所做的研究。该研究结合了14个网络文化与教育专员提供的数据（研究人员、教育工作者、新闻工作者以及企业家）。他们的报告涵盖了基金会对于900名教师的在线调查、对70多名女性的定性焦点研究和对现有研究的评审，为的是对计算机文化、教师观点和课堂互动、教育软件和游戏、计算机科学教室、家庭社区和工作等诸多方面提供深入洞察观点[3]。与Margolis、Fisher和Miller一样，AAUW也发现计算机科学文化对于女性的威慑作用。他们发现女孩都担心她们像工具一样被动地与电脑互动。此外，他们还发现，女孩抵制计算机游戏中的暴力、冗余和沉闷，也对纯技术的编程课程感到反感。此外，AAUW争辩说，这些担忧是因为紧张或是能力不济，一旦技术能力赶上，便会消失。

最后，在一个对IT、CS和CE研究的综合汇编中，McGrath Cohoon 和Aspray集合了该领域34

个主要研究人员的研究[32]。他们对于女生数量少的潜在解释包括：经验、入门障碍、榜样、辅导、师生互动、同学支持、课程和教学方法，以及如学术适合度、价值、信心和如何面对竞争等学生特质，外加计算机文化。

受基于文化的关注启发，我们可能会问，那些没有选择计算机的高能力的女性们选择了什么职业呢？Ceci、Williams和Barnett提醒我们，数学能力强的女性比男性更有可能同时具有较强的语言能力，这种不成比例的优势让她们有更广泛的职业选择权[10]。因此，文化因素和个人选择相叠加，使得有能力的女性离开了计算机领域，从而揭示生理因素和纯粹能力并非唯一的原因。图13-1揭示了这些生理，文化因素的互相影响。

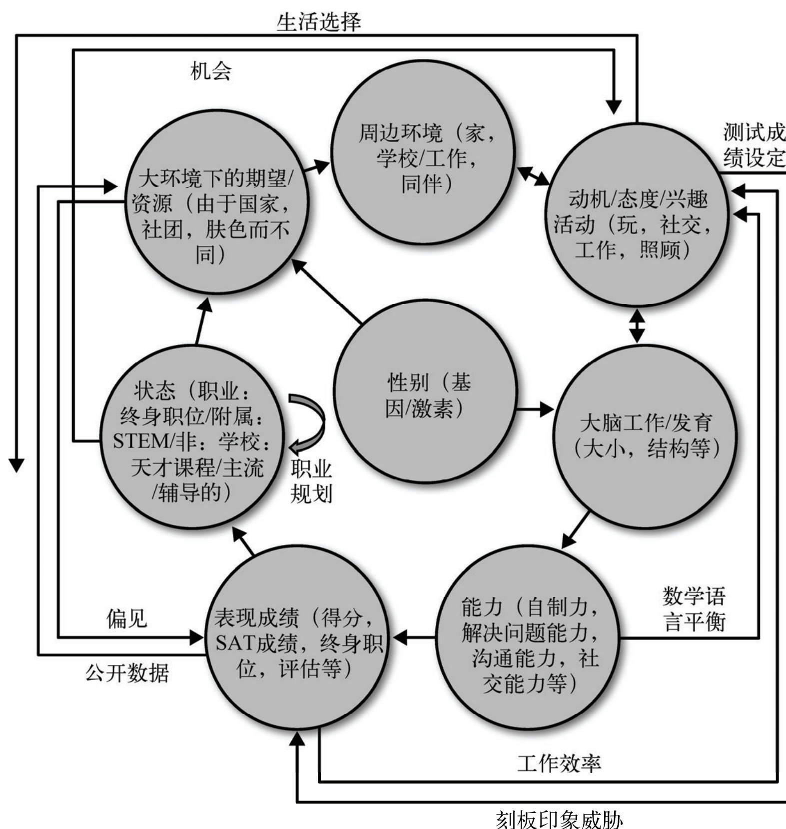


图13-1 科学、技术、工程和数学的性别差异的一般因果模型。图片版权2009由Stephen J. Ceci, Wendy M. Williams，以及Susan M. Barnett所有；授权使用

有这么多的干扰因素，所以毫不奇怪，我们还没有明确的解决办法来解决女性在CS和相关领域可能面临的障碍。另一方面，我们对于多种相互联系、影响女性全面参与的力量有了更多的了解，这给了我们如下启发。

13.2　值得在意吗

在某种程度上，女性不选择计算机科学是因为文化中的困扰因素，而这些因素是可以改变的，但我们要问自己的是应不应该把更多的女性推入计算机科学呢，比如通过教育政策？因为计算机是一个理想的职业，女性有更多机会从事该职业必然会得益。而且，计算机是全球竞争的领域，性别的包容性会带来好处。另外，多样性会改进计算机和软件团队的产品。

然而，这个问题的最终意义可能比任何可立即衡量的益处要大得多。目前研究的不足也许是在建议我们换一个思路：承认生物差异和文化影响的诸多特质是这个复杂方程式的关键因素。

首先，让我们说明女性参与CS的潜在利益。第一，IT职位的薪酬大大超过大多数女性为主的职业[6]。据美国大学与雇主协会的数据显示，2009年7月计算机科学学士学位的毕业生的平均起薪为61 407美元[7]。2008年5月，计算机系统软件工程师的行业最普遍年薪的中位数为：研发部门，102 090美元；电脑及周边设备制造业，101 270美元；软件出版商，935 790美元；电脑系统设计及相关服务，91 610美元。

美国劳工统计局把计算机软件工程师的就业前景评为优秀。展望2008年至2018年，劳工统计局网站预测就业比例的变化为：计算机软件工程师和计算机程序员会增加28.3万个职位，增长21%的；计算机软件工程师会增加29.5万个就业机会，增长32%；软件工程师会有34%的增幅。只有在程序员岗位显示有3%下降。因此，CS是一个具有良好报酬和良好就业前景的新兴领域。

预计到2016年，相对于其他STEM职业，计算机产业将有最大的经济增长和需求（图13-2）。

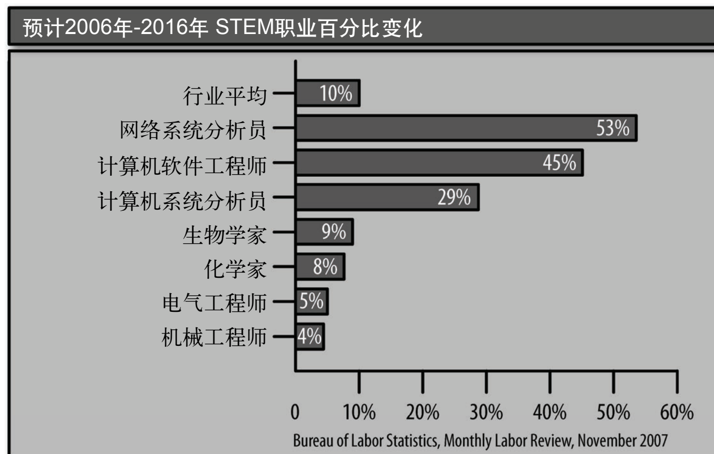


图13-2　预计2006年~2016年 STEM职业百分比变化

技术类工作机会的增长预计比其他专业工作更快，在未来十年涨幅将高达25%[4]。考虑到庞大的需求和就业机会延续到2018年，一个男性为中心的工作文化可能会阻止女性从CS职业中受益。

更多女性参与到计算机行业会得到的经济利益是显而易见的，但除了这些，让女性能参与到

所有专业领域（包括计算机）所带来的益处会更大。美国需要有能力的人从事计算机相关的工作，并且要做好。美国劳工部估计，到2016年，将有超过150万个计算机相关的工作[6]。

尽管美国的科技产业是发展最快的产业之一，按照目前的趋势，到2016年，美国计算机科学本科毕业生只能填补该产业一半的职位[6]。换句话说，无论男女，只要具有潜力和能力的人参与这个行业，都是对社会有益的。

除此之外，性别平衡会有另一些益处，比如多样性。事实上，一些学者曾提出多样性，包括性别多样性，会提高团队表现，虽然不是所有的学者都同意这种说法，因为往往这种说法是基于社会政治基础而不是学术基础。围绕自我分类、社会身份和相似性吸引的研究容易对多样性产生悲观的看法，而信息处理方法却容易得出乐观的结果。正如Mannix 和Neale的解释[29]：

> 围绕自我分类/社会身份和相似性吸引的研究容易对团队中的多样性抱有悲观的看法。在这些模式中，个人会更被其他类似的人吸引，同质团队会更有凝聚力，更加融合。相反，信息处理的角度提供了更为乐观的观点：多样性可以提高团队工作表现。信息处理的观点认为，在相异的群体中，个人会接触到背景、社交网络、信息和技能完全不同的人。这些信息会改善团队的成果，尽管它可能造成该组的协调问题。

Page作为多样化的提倡者，认为在适当的条件下，多样化的团队一贯跑赢由“能力最佳”者组成的团队[34]。基于他在复杂系统、经济学、政治学上的广泛工作成果，Page声称进步不仅取决于个体智商，也同样取决于集体差异。

在对IT工作环境多元化益处的研究表明男女数量相同的团队（比任何其他比率组成的团队）更会尝试、创新、分享知识、完成任务[27]。并且，由男女共同组成的团队所获得的专利，被引用的次数往往比同类型的专利多26%～42%[5]。

关于这一课题的研究往往认为多样性会提高团队的表现，但我们也必须看到，社会学家们50年来的研究结果表明，优势并没有那么明确。Mannix和Neale[29]在报告的第237页指出，任期多样性（雇员任职长度的多样性）对工作表现有不良影响。社会分类上的多样性，比如在年龄、性别和种族因素上的多样性所带来的影响似乎时好时坏，而其结果又与比例直接相关（即少数派与多数派之间的比率）。在一个大规模、包含四个研究的项目中，研究人员测量了种族和性别多样性对于进度和表现的影响。Kochan和同事们发现，性别差异对于团队进度要么没有影响，要么没有积极的影响，而种族多样性往往有负面影响[26]。虽然Kochan和他的同事认为无论哪种多样性，都很少有直接影响，他们也表示环境因素（团队间的竞争）加剧了种族多样性的负面影响。

有趣的是，Sackett和同事提出：在评估多样性的益处时，绩效究竟是如何被评估的[35]。也就是说，作者们意识到绩效考评很棘手。在控制了男性与女性的认知能力、心理能力、教育水平和经验的差异之后，当女性的比例很小时，女性的绩效评级较低。Sackeet和同事们发现，当女性的比率小于20%时，她们的绩效评级低于男性，但当比例大于50%时，她们的评级高于男性。没有在男性的评级中发现任何人数比率和评级的平行关系。由于评估者的性别没有记录，其他解释可能包括对于集体诉讼的恐惧或害怕歧视索赔，这些都难以评估。

换句话说，研究人员缺乏可信的手段来衡量多样性（至少对绩效）的影响。提高女性比例是

否能真正提高绩效，或者是有一些其他潜在因素造成了绩效提高的感觉？怎样研究公开多样性（男、女，黑人、白人）才能同时适当地评估相似和差异的价值和对其的态度？性别、种族不同，但态度和价值观相似的团队是异构还是同构团队呢？显然，需要定义一些参数，制定有效的衡量方法是这方面研究的困难之处。

面对这些困惑，职工多元化的潜在好处之一是财政奖励，这值得注意。2006年Catalyst的一项研究发现，公司董事会成员女性比率越高，公司效益越好。研究声称从净资产收益率、销售和投资资本的回报来说，董事会成员的女性比例高的公司收益超出最低的公司53%、42%、66%[24]。此前，2004年Catalyst的一项研究表明，女性领导比例最高的公司股本回报率比其他公司高35.1%，股东回报高34%。但是，这些结果可能是因为进取心而不是性别。此外，Adams和Ferreira发现性别多样性对市场估值和经营业绩的影响是负面的[1]。这种负面效应，他们解释，可能是由公司股东权利强大造成的。股东权利较弱的情况下，性别多样性有积极的效果。因此，考虑到Catalyst的研究人员无法控制如企业的态度和股东参与这些变量，我们需要质疑其“面值”的结论。

同样要关注的是政治上的强制措施造成的董事会性别多样性。2003年，挪威议会通过了一项法律，要求所有公共有限公司的董事会中，至少有40%是女性。自那时以来，密歇根大学的研究人员就开始调查该法律的后果。Ahern 和Dittmar发现了对公司价值的负面影响，但是，他们很快就指出，造成损失并不是因为新任董事会成员的性别，而是由她们的低龄化和缺乏高层工作的经验[2]。一味关注性别的多样性而使董事会人员的经验降低，至少在短期内，损害了个别公司的利益。这项强制政策的长期结果还有待观察。

最后，有些人认为，一个多元化的员工队伍促进创新。1980年~2005年IT类别中专利申请数量大幅增长，但美国女性的专利增长更为显著。从1980~1985年间，美国所有的IT专利（同时包括两个性别）数量从32 000余个增长到176 000余个，翻了5番[4]。在同一时期，美国女性的IT专利数量从707增长到超过10 000，翻了14番。这一点尤其值得注意，因为IT界就业女性所占比例仍相对固定[4]。此外，女性影响80%的消费者支出决定，但90%的技术产品和服务是由男性设计，这里有一个潜在的未开发的女性市场[19]。技术设计过程中女性的加入，可能意味着市场上会出现更多有竞争力的产品。

W. A. Wulf，是美国国家工程院院长，对于多样性有如下观点：“没有多样性，我们就限制了所应用过的生活经验集合，结果我们会付出机会成本，它们存在于那些未生产的产品，欠考虑的设计，不理解的约束，以及未发明的流程。”另一方面，有关对多样性的研究，麻省理工学院管理与工程教授Thomas A. Kochan说过：“多元化产业的根基是不牢靠的。花言巧语的多元化商业案例是幼稚而做作的。在商业绩效上，性别和种族的多样性没有很强的正面或者负面影响力。”但他也承认，“社会意义上，我们的确需要在所有组织中推广多元化，而且随着时间的推移，当劳动力市场变得更加多元化时，机构绝对需要加强这种能力来保持战斗力[25]。”所以对当前状况最简短的总结是，性别多元化有好处，但也有开销。

13.2.1　扭转这种趋势，我们可以做些什么

关于在CS专业中性别失衡原因的研究，引发了不少充满激情的辩论，提出了对改变的需求。

一些人反驳道，女性选择了她们希望从事的行业，如医学（其中新医学博士中50%是女性）、兽医（其中新兽医中76%是女性）以及诸如生物学领域（其中女性男性人数均衡[8]）。但是，如果社会希望探索鼓励更多的妇女进入计算机领域的可能性，我们可以做什么？计算机界绝大多数是男性的现状能得到扭转么？幸运的是，已经有研究探索了女性在CS领域人数很少的原因，也有了相关的研究来探讨在文化、课程、信息和政策方面的可能干预措施。

在专科教育里，卡耐基梅隆大学的首创研究为CS教学提供了循证干预的优秀范例。这些方法包括把不同背景的学生一起带到跨学科课程中共同处理多元问题，本科教育中对人机交互的关注，以及让学生参与与当地社区非营利组织互动的课程，并运用自己的知识解决社区问题[30]。此外，卡耐基梅隆发现直接录取女性对促进计算机领域的女性参与有很强的影响。通过他们的招生计划和之前所述的课程，他们计算机系女大学生的比例由1995年7%增长到至2000年的40%。尽管2007年全国计算机科学的入学率总体下降，卡内基梅隆是一个例外，该校女生入学率占到23%。

13.2.2 跨国数据的意义

2004年，Charles和Bradley分析了经济合作发展组织（OECD）提供的21个工业化国家高等教育学位授予的数据。正如预期的那样，妇女主要集中在传统的女性型领域，如健康和教育，而在传统男性领域中落后[11]。在所有21个国家中，女性在计算机科学中的任职人数偏低（表13-1）。令人惊讶的是，这组数据包括了男女平等的国家以及男女不平等国家，人们可能预期在男女不平等的国家里，女性人数的不足（或男性比例过高）将会是最严重的。然而，土耳其和韩国，都是男女不平等的国家，阳盛阴衰的情况却较小（见表13-1）。这可能部分因为政策规定计算机行业男女都要参与的缘故。阳盛阴衰指数表示每个国家计算机领域男性超过女性的倍数（数据如何得出请参照Charles和Bradley的报告[11]。

表13-1 2001年，计算机科学专业中“阳盛阴衰指数”*

国 家	阳盛阴衰指数	国 家	阳盛阴衰指数
澳洲	2.86	荷兰	4.39
奥地利	5.37	新西兰	2.92
比利时	5.58	挪威	2.75
捷克	6.42	斯洛伐克	6.36
丹麦	5.47	西班牙	3.67
芬兰	2.29	瑞典	1.95
法国	4.57	瑞士	4.66
德国	5.58	土耳其	1.79
匈牙利	4.66	英国	3.10
爱尔兰	1.84	美国	2.10
韩国	1.92		

* 值表示了每个国家男性阳盛阴衰指数。这个值是之前计算中计算机科学参数（参见第6章中McGrath Cahoon and Aspray, 2006 以及Charles和Bradley的论文[11]）的倒数，并且把结果的正值转换为指数形式。

Charles和Bradley的研究并不符合社会进化理论，因为经济最发达国家计算机科学领域的女性并没有更多。同样，作者表明，在职女性数量或者高职位女性数量与学习计算机科学的数量并没有很强的相关性。这些结果再次表明，计算机专业女性人数不足的原因更像是文化因素，而不是生理因素，而文化因素是可以改变的。但需要着重注意的是，研究并没有提供证据证明，经济最发达的国家计算机科学中的女性更多，或是劳动力市场、高等教育或高地位专业中女性多的国家有更多的女性在计算机科学领域[11]。因此，女性的偏好成为了女性就职现象最可能的解释，而不是暗示性偏见阻止了女性进入计算机行业。

所有21个国家计算机科学女性人数的不足代表了在这些国家的文化中有一个强烈的信念，即男性和女性的分工有所不同。对于Charles和Bradley的研究，最有意思的地方在于，国家与国家之间有着很大的差别，这就意味着社会文化因素的影响非常之大。在美国，我们强调教育的社会目标是培育自由选择和自我实现，然而主流社会的各种成见又可能暗中扼杀了学生的“自由”选择，因为他们可能会去追求传统文化中应由男性或女性担任的工作。Charles和Bradley指出，政府对课外课程严格管制的国家（如韩国和爱尔兰）中，计算机科学相关行业女性人数不足的情况较轻。这表明，我们可能要推迟青少年的职业选择到他们不太会受性别成见所影响的时候，并落实各种政策，让学生从幼儿园到12年级以及之后都有机会探索数学和科学，包括计算机科学。

13.3 结论

在这一章中，我们提供了最新的证据，帮助读者浏览和探索为什么很少有女性从事计算机职业这一问题，为什么我们应该关心这个问题，以及如果我们可以为此做些什么的话，该如何做。我们提到男性女性之间的生理差异以及认知能力差异，尤其体现在极具天赋的个人上；在职业生涯和生活方式上的偏好差异；以及计算机科学环境文化。尽管在理解性别和CS/IT领域参与人数的关系时，有明显的鸿沟，基于关于女性在科学领域的实证文献，仍有必要讨论应不应该鼓励更多女性参与CS领域并权衡其代价。

总之，产业界和商业界的一些人认为，CS/IT相关领域缺少女性是不利于女性经济地位和全国经济发展的，而另一些人持相反观点。尽管一些关于CS领域女性数量不足的跨国比较[11]质疑了干预措施的意义，总的来说，如果政策制定者能在学生年龄尚小，还没有被性别身份角色左右之前，就推进男女学生接触计算机，似乎会是一个明智之举。考虑到对于女性和社会的潜在好处，可以考虑鼓励女性进入信息技术、计算机科学和计算机工程领域的步骤。很多研究者已经提出了文化、课程、信心有关的干预措施[30] [32] [3]，首先需要持续评估它们是否有效，能帮助还是阻碍女性参与计算机科学领域，以及这些变化是否真正地改善了这个领域。最终目标应该是计算机专业的质量、有效性和发展，无论这是否意味着CS未来大部分是男性、女性、或是平衡的性别组成。

13.4 参考文献

[1] [Adams and Ferreira 2009] Adams, R., and D. Ferreira. 2009. Women in the boardroom and their impact on governance and performance. *Journal of Financial Economics* 94(2): 291-309.

[2] [Ahern and Dittmar 2009] Ahern, K., and K. Dittmar. 2009. The changing of the boards: The value effect of a massive exogenous shock. Under review.

[3] [AAUW 2000] American Association of University Women. 2000. *Tech-Savvy: Educating Girls in the New Computer Age*. Washington, DC: American Association of University Women Educational Foundation. Retrieved from *http://www.aauw.org/research/upload/TechSavvy.pdf*.

[4] [Ashcraft and Blithe 2009] Ashcraft, C., and S. Blithe. 2009. *Women in IT: The Facts*. Boulder, CO: National Center for Women & Information Technology.

[5] [Ashcraft and Breitzman 2007] Ashcraft, C., and A. Breitzman. 2007. Who invents IT? An analysis of women's participation in information technology patenting. National Center for Women & Information Technology. Retrieved from *http://www.ncwit.org/pdf/PatentExecSumm .pdf*.

[6] [Bureau of Labor Statistics 2004] Bureau of Labor Statistics. 2004. *Occupational Outlook Handbook*, 2004-05 Edition. Washington, DC: Labor Department, Labor Statistics Bureau. Retrieved from *http://www.bls.gov/oco /ocos267.htm*.

[7] [Bureau of Labor Statistics 2010] Bureau of Labor Statistics. 2010. *Occupational Outlook Handbook*, 2010-11 Edition. Washington, DC: Labor Department, Labor Statistics Bureau. Available at *http://www.umsl.edu/ services/govdocs/ooh20042005/www.bls.gov/OCO/index.html*.

[8] [Ceci and Williams 2010] Ceci, S.J., and W.M. Williams. 2010. *The mathematics of sex: How biology and society conspire to limit talented women and girls.* New York, NY: Oxford University Press.

[9] [Ceci and Williams 2007] Ceci, S.J., and W.M. Williams, ed. 2007. *Why aren't more women in science? Top researchers debate the evidence*. Washington, DC: American Psychological Association Books.

[10] [Ceci et al. 2009] Ceci, S.J., W.M. Williams, and S.M. Barnett. 2009. Women's underrepresentation in science: Sociocultural and biological considerations. *Psychological Bulletin* 135(2): 218-261.

[11] [Charles and Bradley 2006] Charles, M., and K. Bradley. 2006. A matter of degrees—Female underrepresentation in computer science programs cross-nationally. In *Women and Information Technology: Research on Underrepresentation*, ed. J. McGrath Cohoon and W. Aspray, 183-204. Cambridge, MA: MIT Press.

[12] [College Board 2008] College Board. 2008. *AP summary report* (*calculus AB, computer science A and AB*). Princeton, N.J.: College Board. Retrieved from *http://professionals.collegeboard.com/data-reports-research/ ap/archived/2008*.

[13] [Deary et al. 2007] Deary, I., K. Ferguson, M. Bastin, G. Barrow, L. Reid, J. Seckl, J. Wardlaw, and A. MacLullich. 2007. Skull size and intelligence and King Robert Bruce's IQ. *Intelligence* 31: 519-525.

[14] [Durndell and Lightbody 1993] Durndell, A., and P. Lightbody 1993. Gender and computing: Change over time? *Computers and Education* 21(4): 331-336.

[15] [Eccles et al. 1999] Eccles, J., B. Barber, and D. Jozfowicz. 1999. Linking gender to educational, occupational and recreational choices: Applying the Eccles et al. related model of achievement related choices. In *Sexism and stereotypes in modern society: The gender science of Janet Taylor Spence*, ed. W. Swan, J. Langlois, and L. Gilbert, 153-192. Washington, DC: American Psychological Association Books.

[16] [Ferriman et al. 2009] Ferriman, K., D. Lubinski, and C. Benbow. 2009. Work preferences, life values, and personal views of top math/science graduate students and the profoundly gifted: Developmental changes and gender differences during emerging adulthood and parenthood. *Journal of Personality and Social Psychology* 97(3): 517-532.

[17] [Haier et al. 2005] Haier, R., R. Jung, R. Yeo, and M. Alkire. 2005. The neuroanatomy of general intelligence: Sex matters. *Neuroimage* 25(1): 320-327.

[18] [Halpern et al. 2007] Halpern, D., C. Benbow, D.C. Geary, R. Gur, J. Hyde, and M.A. Gernsbacher. 2007. The

science of sex differences in science and mathematics. *Psychological Science in the Public Interest*, 8(1): 1-52.

[19] [Harris and Raskino 2007] Harris, K., and M. Raskino. 2007. *Women and men in IT: Breaking sexual stereotypes.* Stamford, CT: Gartner.

[20] [Hedges and Nowell 1995] Hedges, L., and A. Nowell. 1995. Sex differences in mental test scores: variability and numbers of high-scoring individuals. *Science* 269: 41-45.

[21] [Higher Education Research Institute 2008] Higher Education Research Institute. 2008. *Freshman: Forty year trends, 1966-2006*. Los Angeles, CA: Higher Education Research Institute.

[22] [Honey 1994] Honey, M. 1994. The maternal voice in the technological universe. In *Representations of Motherhood*, ed. D. Bassin, M. Honey, and M.M. Kaplan, 220-239. New Haven: Yale University Press.

[23] [Hyde and Lynn 2008] Hyde, J., and M. Lynn. 2008. Gender similarities in mathematics and science. *Science* 321: 599-600.

[24] [Joy and Carter 2007] Joy, L., and N. Carter. 2007. The bottom line: Corporate performance and women's representation on boards. *Catalyst Research Reports*. Retrieved January 29, 2009, from *http://www.catalyst.org /publication/200/the-bottom-line-corporate-performance-and-womens-representation-on-boards*.

[25] Kochan 2010] Kochan, T. 2010. Personal communication, March 16.

[26] [Kochan et al. 2003] Kochan, T., K. Bezrukova, R. Ely, S. Jackson, A. Joshi, K. Jehn, J. Leonard, D. Levine, and D. Thomas. 2003. The effects of diversity on business performance: Report ofthe diversity research network. *Human Resource Management* 42:3-21.

[27] [London Business School 2007] London Business School. 2007. Innovative potential: Men and women in teams. Available at *http://www.london.edu/newsandevents/news/2007/11/Women_in _Business_Conference_725.html*.

[28] [Lubinski et al. 2001] Lubinski, D., C. Benbow, D. Shea, H. Eftekhari-Sanjani, and B. Halvorson. 2001. Men and women at promise for scientific excellence: Similarity not dissimilarity. *Psychological Science* 12: 309-317.

[29] [Mannix and Neale 2005] Mannix, E., and M. Neale. 2005. What differences make a difference? The promise and reality of diverse teams in organizations. *Psychological Science in thePublic Interest* 6(4): 31-55.

[30] [Margolis et al. 2000] Margolis, J., A. Fisher, and F. Miller. 2000. The anatomy of interest. *Women's Studies Quarterly* 28(1/2): 104.

[31] [Martin 1992] Martin, C., ed. 1992. *In search of gender-free paradigms for computer science education*. Eugene, OR: International Society for Technology in Education.

[32] [McGrath Cohoon and Aspray 2006] McGrath Cohoon, J., and W. Aspray, ed. 2006. *Women and Information Technology: Research on Underrepresentation*. Cambridge, MA: MIT Press.

[33] [National Center for Education Statistics 2008] National Center for Education Statistics. 2008. *Classification of Instructional Program 11*. Washington, DC: U.S. Department of Education Institute of Education Sciences.

[34] Page 2007] Page, S. 2007. *The Difference: How the power of diversity helps create better groups, firms, schools, and societies.* Princeton, NJ: Princeton University Press.

[35] [Sackett et al. 1991] Sackett, P., C. Dubois, and A. Noe. 1991. Tokenism in performance evaluation: The effects of work group representation on male-female and White-Black differences in performance ratings. *Journal of Applied Psychology* 76(2): 263-267.

[36] [Schofield 1995] Schofield, J. 1995. *Computers and classroom culture*. New York: Cambridge University Press.

[37] [Seymour and Hewitt 1994] Seymour, E., and N. Hewitt. 1994. *Talking About Leaving: Factors Contributing to High Attrition Rates Among Science, Mathematics, and Engineering Undergraduate Majors.* Boulder, CO: University of Colorado, Bureau of Sociological Research.

第14章

两个关于编程语言的比较

Lutz Prechelt

在聚会中，典型的程序员们往往很安静。不过有那么一个话题，他们不仅会全神贯注地听，而且会积极地参与到讨论中来，那是什么话题呢？编程语言！每个程序员都有很多关于各种编程语言的故事可说，对于哪种语言是最好的以及为什么，也有着自己的许多看法。有证据支持吗？当然。经验丰富的程序员们还会有许多的“实战故事”，来告诉人们某种编程语言会导致事情往好或者不好的方面发展：“我本来估计这是个十小时或十五小时的活，但后来我转用X语言，把所有的代码写完让它跑起来只用了三个小时。而且代码的可读性竟然也不错！”

但这种证据有个问题，它通常不包括任何对于编程语言的直接比较。如果其中存在比较的话，有类似于苹果与橘子之间的比较就算是不错的了，大部分情况下更像是苹果与猩猩之间的比较。

人们认为科学家们会抓住这个机会，为计算机编程领域做出广为人知的巨大贡献，并且会产出一系列出色而又清晰的研究来比较各种编程语言的优缺点。有大量课题可以候选：（语言）执行速度、内存消耗、缺陷率、缺陷类型、可靠性、稳健性、可读性、可修改性、编程效率等。

但我也不知道为什么，到目前为止这种情况从未发生过。这方面的科学证据非常不足。此外，尽管聚会中的实战故事远比严肃的科学研究吸引人，但这些故事的可信度并不高[4]。

在我的职业生涯中，我曾遇到过两次机会对编程语言进行合理的有说服力的比较，而且我抓住了它们。本章讲述的就是这两次研究的故事。

14.1　一个搜索算法决定了一种语言的胜出

我曾为一项研究收集了许多对同一程序的不同代码实现，第一个机会就此产生。那项研究[10]评估了Watts Humphrey的个人软件过程（PSP[5]）的培训效果。PSP号称培训之后，能在估算准确度、缺陷密度和生产力方面有较大的提高，但我们的研究发现，效果比预期小很多。

参与该项研究的研究生中一半接受了PSP的培训，其余则接受了其他编程相关的培训。在该项研究中，他们都被要求解决同样的任务（在下一节中介绍），但可以自由地选择编程语言。最

后产生了40种不同的代码实现（用Java语言的有24个，用C++的有11个，用C语言的有5个）。这时我发现如果从三种语言的角度来比较程序，会比单纯比较是否接受过PSP培训的学员更有意思。如果可以进一步比较用不同脚本语言实现的程序，那会更有意思。

于是，我在几个Usenet的新闻组里面发布了召集编程实现的帖子（那是1999年），在四周之内我就收到了另外40个编程志愿者们寄过来的代码：用Perl实现的有13个，用Python的有13个，用Rexx的有4个，用TCL的有10个。

这时，你会质疑说："等等，你怎么知道这些人的编程能力有可比性呢？"很好的问题，我们会在最后讨论可信度时回到这个问题上来。

14.1.1　编程任务：电话编码

程序员要实现下面的程序。

程序首先加载一个字典到内存里（一个纯文本文件，每行有一个单词，整个测试文件中包含73 113个单词，一共938KB）。接着，它从另一个文件读取"电话号码"（由任意数字、破折号和斜杠组成的最大为50个字符的长字符串），把它们一个一个转成词序列，并打印出结果。字符和数字之间的转换有预定义的固定映射（旨在平衡数字频率），如下所示：

```
e jnq rwx dsy ft am civ bku lop ghz
0 111 222 333 44 55 666 777 888 999
```

程序的任务是要找到一列词，这些词中的字符序列正好对应电话号码中的数字序列。必须找到所有的对应结果，并打印出来。答案是由一个一个词组建而成的，如果在这个过程中，字典中没有一个词可以插入某个位置，那就可以用电话号码中这个位置的数字插入。许多电话号码压根没有任何对应结果。

这里有一个电话号码`3586-75`的例子。字典里面含有`Dali`, `um`, `Sao`, `da`, 以及`Pik`这些词。

```
3586-75: Dali um
3586-75: Sao 6 um
3586-75: da Pik 5
```

在处理每一个号码的时候，程序需要保存中间转换的结果。这就需要程序把字典读入一个高效率的数据结构中，因为这里我们严禁为每个需要解码的数字扫描整个字典。

程序员必须保证实现是百分之一百可靠的。他们有一份小字典用来进行测试，以及一组输入输出作为例子。输入的是电话号码，输出的是这些号码对于给定字典的所有正确译码。

14.1.2　比较执行速度

图14-1显示了7种语言运行时间的比较。（参考下面的"如何阅读盒状图"来理解盒状图格式。）

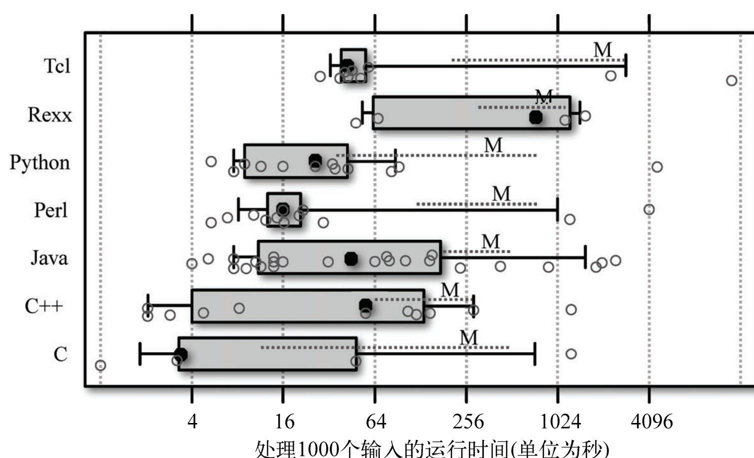


图14-1 每种语言处理1000个输入的运行时间。坏结果对好结果的比率从Tcl的1.5到C++的27。请注意，数轴上的数字是对数增长的，拿Java来说，运行时间的差距可以达到500倍

如何阅读盒状图

盒状图是一种很好的可视化工具，它可以用来很快地比较不同数据样本的位置与分布情况。在我们这个盒状图中，每一个小圆圈表示一个数据点。

盒子表示数据的“中间的一半”，即盒子的左边缘代表了25百分点，即25%的数据点小于或等于这个点（必要时会加上插补值）；盒子的右边缘代表75百分点。

胖圆点表示50百分点（中位值）。为了跟中位值区别开来，平均数用字母M标明。M周围的虚线代表着平均值加减一个标准误差之后的范围，这表明在当前样本中，人们在68%的置信度下估计总体平均值的准确度。

盒子左右的短竖线分别表示10%和90%的分界点，比用最大最小数更好地表示了极值。盒子很宽或者虚线很长意味着数据的变动性很高，也就是说不确定性很大，而对于软件开发来说，就是高风险的意思。我们用盒子右边缘比左边缘的比率来衡量变动性，叫做好坏比率，或者坏好比率。左边缘是数据下半部分的中值，如果数据是记录消耗的话，那下半部分是“好”的数据，因为消耗总是越少越好。上半部分同样如此。

当用可视化比较编程语言时，我们可能会比较胖圆点和M点（平均值）的位置，或整个盒子。盒子的形态能最明显地表现出不确定性，这降低了我们得出仓促结论的可能。

比可视化比较工具更可靠（但欠明显）的比较手段是基于统计的数值比较。在目前情况下，可以通过随机引导方式计算置信区间。请注意，有时数据相差十分大，图上需要使用对数刻度，这会阻碍你的想象力。想想清楚吧！

此外，C语言和Rexx语言的图有点不可靠，因为它们分别只有四五个实现。

记住，这些数据是对数刻度的，这意味着图中同语言内的差异是巨大的。它大到21个语言平均运行时间的成对差异中，没有一个具有统计显著性。这点可以从图中每种语言的虚线都高度重叠看出。这意味着不能忽视这种可能性（显然超过5%，即统计显著性的临界点）：观察到的差异可能仅仅是因为随机波动造成的，因此是不“真实”的。

换句话说，我们可以说程序员之间的差异大于语言之间的差异。有一些运行时间中位数的明显差异（Perl和Python都比Rexx和Tcl更快），但中位数并不很相关，因为你只有在关心一个程序是否比另一种更快而不是快多少的时候才会使用它。

为了从数据中得出更多信息，我们把语言分为三组：Java一组，C和C++一组，所有四个脚本语言一组。在这种分类下，统计分析表明，在80%的置信度下，脚本程序运行时间至少是C / C++程序的1.29倍，Java程序运行时间至少是C / C++的1.22倍。这没什么大不了的。这时我会听到你说：“我不相信！C和C++远远要比脚本语言快**无数倍**。这些差异到底跑到哪里去了？”

这个问题有两个答案。首先我们把运行时间分为两部分。该电话编码程序执行有两个阶段：第一，加载字典和建立相应的数据结构；二、对于每个输入的号码执行搜索。在搜索阶段，脚本语言的内部差距相当大，Java的内部差距非常大，而C / C++的内部差距则巨大无比。编程语言内部的差别是如此之大，以至于语言之间呈现的差异在统计意义上变得微不足道。不过，在加载字典阶段，C / C++程序确实是最快的。Java程序至少需要平均1.3倍的时间才能完成，而脚本程序至少需要平均5.5倍的时间（80%的置信度）。

第二个答案在于编程语言内的差距，这是令人难以置信的，特别是对C和C++而言。这很大程度上模糊了语言间的差异。还记得之前提过的好坏比率么：C ++程序中较慢一半运行时间的中位值是较快一半中位值的27倍。难道开发较慢一半的程序员都是大白痴？不，他们不是。下面关于程序结构的讨论会揭开到底这些差异是从哪里来的。

14.1.3　内存使用情况的比较

与上一节中运行的程序相同，图14-2显示了不同语言实现的内存使用情况比较。

从图中，我们观察到：

- 稍小的C / C++程序内存消耗最低；
- C++和Java明显分成两个小团体（看小圈），消耗大的与消耗小的；
- 脚本语言往往不如C / C++；
- Java语言平均来说明显比其他语言都差；
- 对于内存使用来说，用哪种语言都可能让你吃不了兜着走；
- Java的差异性最大，所以计划风险最大。

接下来的分析里面，我们会讨论把程序分段的必要性。

14.1.4　比较效率和代码长度

在千兆内存和千兆赫兹、多核CPU的时代，运行时间和内存消耗往往并不重要。但是，如何

帮助程序员省力总不会过时。图14-3显示了程序员们编写每个代码实现所花时间，可能这是本次研究中最壮观的结果了。

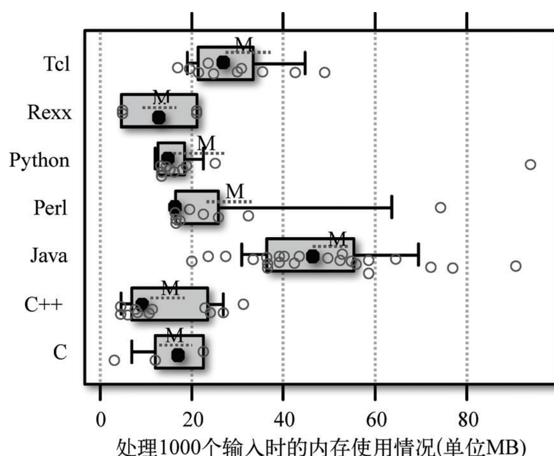


图14-2 处理1000个输入时的内存使用情况，包括运行中的系统、程序代码和数据使用情况。坏好比率从Python的1.2到C++的4.9不等

我们看到，脚本语言平均用时差不多3至5个小时，非脚本语言却用时9至16小时。非脚本语言大概是脚本语言的三倍。我几乎可以看到所有非脚本语言的程序员都在座位上惴惴不安。这些数据可信吗？或者有许多脚本语言程序员谎报了他们的工作时间？图14-4提供的证据表明，数据是可信的，因为脚本程序长度是非脚本程序的三分之一。不过，解释这些数据时，请参阅14.1.7节的注意事项。

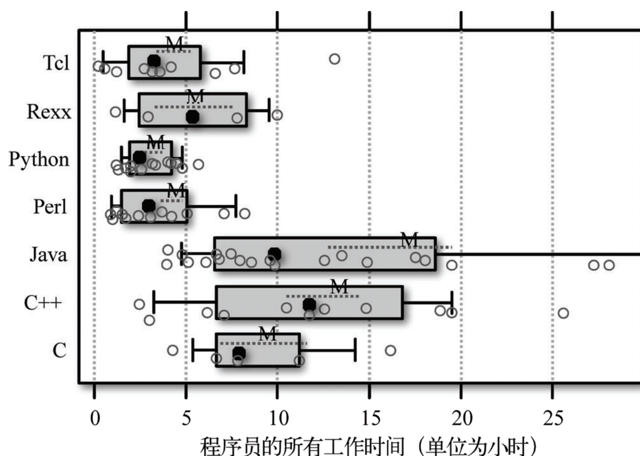


图14-3 程序员的所有工作时间。脚本语言的数据由程序员自己测量和汇报，非脚本语言的数据由实验者测量。坏好比率范围从C的3.2到Perl的1.5。有3个分别是40，49和63小时的Java程序工作时间，在这里没有显示

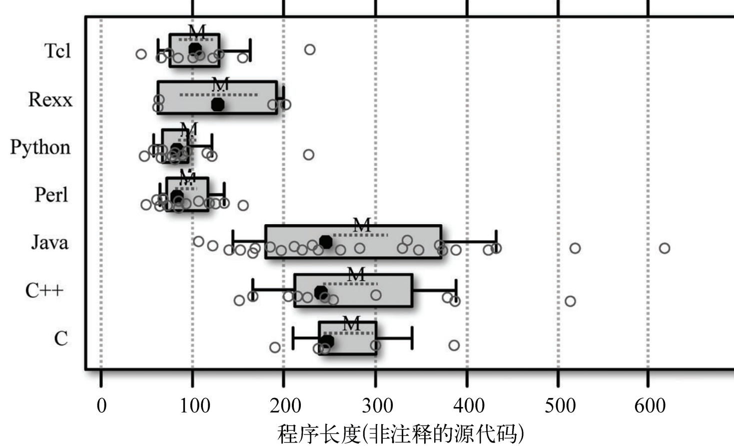


图14-4　程序长度，非注释的源代码行数。坏好比率范围从C的1.3到Java的2.1和Rexx的3.7。脚本语言的评论密度（未显示）比非脚本语言的高

14.1.5　比较可靠性

在对所有程序的可靠性测试中，我们发现大部分程序表现完美。一些为数不多的错误在每种语言都有，至少在考虑脚本语言程序员和非脚本语言程序员不同的工作条件时[6]。此外，在所有语言中某种输入都容易出错：只有破折号和斜线而没有数字的狡猾“电话号码”。总之，实验没有发现语言间表现出的一贯的可靠性差异。

14.1.6　比较程序结构

当我们深入程序的源代码想要查看它们的设计时，我们发现了这个实验的第二个真正精彩的结果。

程序中使用了三种基本数据结构中的一种来表现词典内容，以加快搜索。

- ❑ 根据第一个字母对应的数字，把词典简单划分为10个子群。这是一个简单但也非常低效的解决方案。
- ❑ 一个用电话号码作为索引的关联数组（哈希表），映射到词典里面所有对应的词。这个方案编程上没有增加难度，但非常有效。
- ❑ 一个10元树，其上任何路径对应一个数字序列，词典单词位于所有叶节点和一些内部节点。代码量很大，但是搜索极快。

观察结果很有意思：第一，所有的C，C++和Java程序用了算法1或算法3，这两种方案之间的差异解释了图14-1中这三种语言巨大的语言内差异；第二，所有的脚本都使用算法2。

算法2可以说是最好的解决办法，因为它代码工作量小，效率相当高。但是，尽管C++和Java本身提供了合适的哈希映射实现，但C++和Java的程序员显然没有想到使用它们。相比之下，脚

本语言程序员似乎习惯用哈希表，所以他们都想出了相应简单和紧凑的搜索方法。

使用的语言塑造了程序员的思维，尽管他们中的许多人也了解其他类型的语言。这个结果也解释了有关语言间程序长度的差异：使用关联数组的循环代码量少。

我不认为这些结果和语言的“好”或“差”相关。特定的语言使得程序员倾向于特定的解决方案风格。哪个更好取决于手头的具体问题。

14.1.7　我可以相信吗

请记住，这里我们仍然有一个挥之不去的问题：

> 我们怎么知道，我们比较的这些不同语言的程序员们编程水平相当？

我们可以用上一节中所使用的比较工作效率的方法：比较每小时产出的代码行数。众所周知，每行代码的工作量取决于程序需求和各种制约因素，但几乎跟编程语言没有关系[1]，因此较高的语言表述能力是提高生产力的源泉。因此，如果语言间每小时代码行数都差不多的话，我们有理由相信，每种语言的程序员们是可以互相比较的（大不了会有少数脚本程序员的工作速度快得过分了）。

如图14-5所示，结果中顶多有四个效率值高得离谱（TCL三个，Perl一个），其他所有都落在PSP实验中被监控的Java、C和C++程序员们的数据范围之内，所有语言的平均数都比较相似。这意味着它们可以相互比较，脚本语言程序员报告的工作时间也是大致正确的。

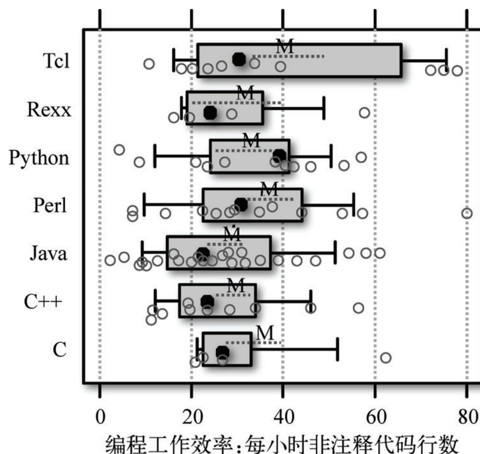


图14-5　编程工作效率：每工作小时非注释行代码的行数。C、C++和Java工作时间是被记录下来的，脚本语言的工作时间是程序员自己上报的

接下来，如果我们同意接受以上的结论，当我们解释这些数据时，仍有三条限制条件需要我们铭记于心。

- 这些度量比较旧（在1999年完成）。现在每种语言或多或少都进行了优化，重新来做实验的话可能会不同。特别是Java（当时的版本1.2）和Python（当时版本1.5）应该已经提高

了很多。此外，在数据采集时，Java还是一门很新的语言，程序员们对它的掌握度还不高。例如，有的程序员没有用`StringBuffers`而是用了Strings了，这解释了Java程序中内存消耗、运行时间中一些不理想的数据。

- 本研究的代码任务小，而且相当具体。其本质也相当特殊。不同的任务可能会带来不同的结果，但我们不能确定。
- 脚本程序员的工作条件与Java、C和C++组的不一样。后者是在实验室条件下，而前者在非实验条件下。特别是一些脚本程序员报告说，他们并没有在读了任务说明之后马上开始写代码，这意味着他们可能有更多的时间进行思考，但这没有在时间测量内体现出来，这是一个优势。幸运的是，在工作时间的差异是巨大的，就算是非脚本语言的工作时间打个很大的折扣，剩下的差异仍旧很大。

其他一些因素也可能影响本研究的结果。如果你想了解，可以去看这个研究的原始数据[6]。也有一个短些的报告[7]，但仍比在这里描述得详细些。

14.2　Plat_Forms：网络开发技术和文化

第二个比较语言的机会是这样产生的：2005年一家网络应用软件服务公司的所有者Gaylord Aulke打电话给我，他说他读了我那篇关于“电话编码”的报告，然后解释给他的客户听为什么不必惧怕使用脚本语言。但他想要一些更具体的东西，于是问我是不是可以为PHP也做个类似的实验呢？

听上去很有意思，但我想了两秒钟后就对他说我做不了，并解释了原因：首先，为了可信，比较网页开发平台需要一个更大任务（至少需要几天时间来做一个看得到的网络应用）；需要团队（至少三个人，这样我们才能研究Web开发的各个方面），而不是像语言组里面一个人完成一个程序；需要高质量的专业选手，而不是随机的人员或是学生（网络开发平台的实际运用需要更深入的知识）。同时，评估方法会更加复杂。我想不出来有谁会给如此庞大的实验买单。

他的反应是“嗯，我了解了。我会找些人，也许我们可以做些什么。”几个月后，在他的努力下，我与Richard Seibt和Eduard Heilmayr取得了联系，这两个人最终吸引了很多团队加入到这个实验中来。我把这个实验计划缩减了相当一部分，命名为Plat_Forms。这个实验的目的是为了寻找在平台内一致、而在平台间不同的解决方案或者开发流程中的一部分或所有特点。这些特点被认为是平台的突出特点，也就是说，在设计或决定采用平台时没有考虑到，但却对开发造成了一贯影响的因素。

除了没有足够的团队做.Net，Python和Ruby的开发之外，在竞赛的名义下采集数据很顺利，我们只有Java, Perl和PHP的。评估过程很困难，不过最终我们也发现了有趣的结果。

14.2.1　开发任务：人以类聚

在2007年1月25日上午，九支过硬的专业队伍（三支使用Java，三支使用Perl，三支使用PHP）分别拿到了20页文件（详细描述了127个详细的功能需求，19个非功能需求，5个组织需求）和光

盘上的两个数据文件。第二天下午，他们递交了一张DVD，包含源代码、版本压缩包、以及包含可以运行解决方案的VMware虚拟机。在此期间，他们可以以任何适合他们的风格工作（甚至包括通过公共原型服务器和博客评论进行现场测试），使用一切可用的软件和他们需要的工具。

功能需求以用户用例的格式表示，并且标有优先级（一定要做，应该做，可以做）。功能需求描述的是一个社区门户网站，叫做“人以类聚”（PbT）。程序需要做到以下几个方面。

(1) 用户注册，其中包括不常见的属性，如GPS坐标。

(2)“简单的气质测试”（TTT），一份包含40个二选一问题的问卷（由光盘上的一个结构化文本文件提供），得出一个人格类型的四维分类。

(3) 搜索用户，基于17个不同的搜索条件（可互相组合），其中一些比较复杂，比如选择16个个性类型的子集，或基于GPS坐标做粗略的距离分类。

(4) 一个用户列表，用于表示搜索结果，或者“与我有联系”的用户群等。其中包括列表的可视化概要，需要用符号将用户在一个基于两个可选的笛卡儿坐标系中表示出来。

(5) 用户状态页面，显示有关用户的详细资料（一些属性只在一定条件下可见）。完成一项协议，使用户能够通过发送和回答“获取联系详情”的请求（RCDs），使他们的电子邮件对对方可见（“取得联系”）。

(6) 另外提供一个基于SOAP的Web服务接口，要对应CD中WSDL文件。

非功能性需求包括可扩展性、持久性、编程风格、一些用户界面的特点，等等。

参与者可以要求澄清问题，但最终每组在每小时中问了不到两次。这项任务被证明是比较有趣且比较适合作对比的，但对给定的时间而言，任务量有点大。

14.2.2 下注吧

作为科学家，我对于实验会发现什么完全没有想法，对于三个平台的特点也没有偏见。你相信吗？可笑至极！我确实是中立的（在并非强烈支持某一种的意义上），但是我理所当然会有一些期望。如果许多期待都被粉碎的话，也是充满了娱乐性和启发性的。为了在这章节结束之前获得最大的乐趣和知识，我鼓励你也想一下你有哪些期待。就现在。我是认真的。

下面是我的大致预测。

- 工作效率

  PHP和Perl较高。
- 源代码长度

  Perl最少，PHP较低，Java最高。
- 执行速度

  Java最快。
- 安全性

  Java最好，PHP最差。
- 构架风格

  Java的干净，Perl和PHP实际（别管这意味着，我自己也不确定）。

- 开发风格

　Java较少使用增量开发，较多使用前期设计。

这些期望中的一部分或多或少地成真了，但其他的却大大出乎意料。

14.2.3　比较工作效率

由于所有的团队的工作时间是一样的（两天），并且依照同样一套细粒度的需求工作，而且没有一支团队提前完成，我们通过计算有多少需求被实现且可用计算工作效率。结果显示在图14-6。

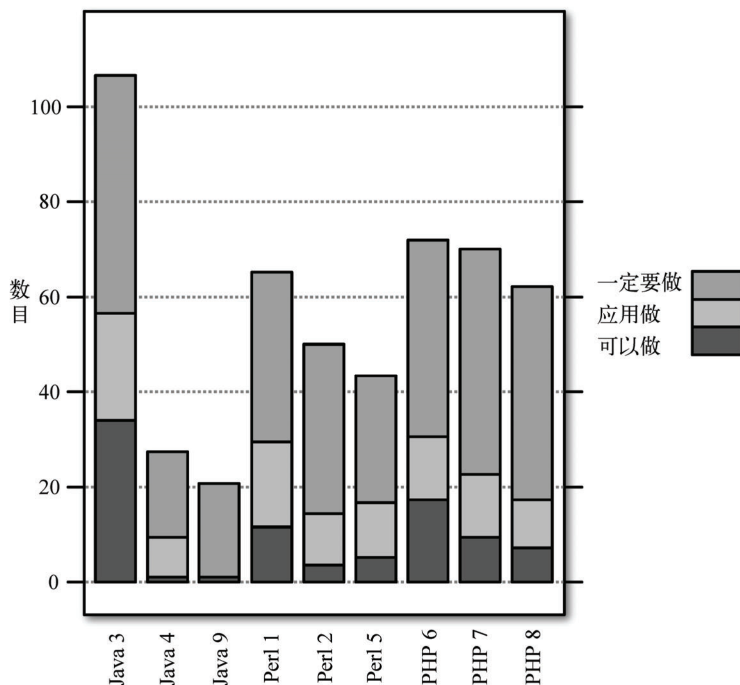


图14-6　在规定时间内，每组完成的需求的数目，以优先级区分（一定要做的，重要的，可选）

请注意，第9组应该在这次评估中被忽略。他们在初期测试版上用了一个不成熟的框架，其局限性使得他们完成不了多少。他们之前就非常不愿意参加，但我们不得不激励他们，因为我们没有其他选择作为第三个Java团队了。第4组花了很多时间在VMware安装上面，与Java没有任何关系。尽管如此，该数据仍然包含了如下有趣的内容。

- ❑ 最有效率的团队恰恰来自最不被看好的Java平台。我们只好猜测一下为什么会如此。一些非第3组的参与者指出只有第3组用了商用的框架。也许正是这个区别解释了这组的最高工作效率，又或许是因为这个团队有良好的“公司风格”，所以对所用的框架最为精通。
- ❑ PHP具有最高的平均工作效率。

- PHP也具有最均匀的工作效率。如果这是一个系统效应的话，这将意味着PHP项目（至少对能力相当的团队来说）是最低风险的。

对于最后一项发现，我从来都没有想到过，但它显然需要进一步研究，因为风险是一项非常重要的项目特征[3]。现在回想起来，我会期望有较高规范的静态语言比动态语言有更平稳的工作效率（统计意义上）。

14.2.4 比较软件工件的大小

虽然各组项目实现的需求数量差别很大，但这与他们源代码的大小差异相比算不了什么。图14-7描述了每组提交的软件大小（代码行数）的大概情况，只有程序源代码和模板文件被算在内（不包括数据文件，二进制文件，编译文件或项目文档）。

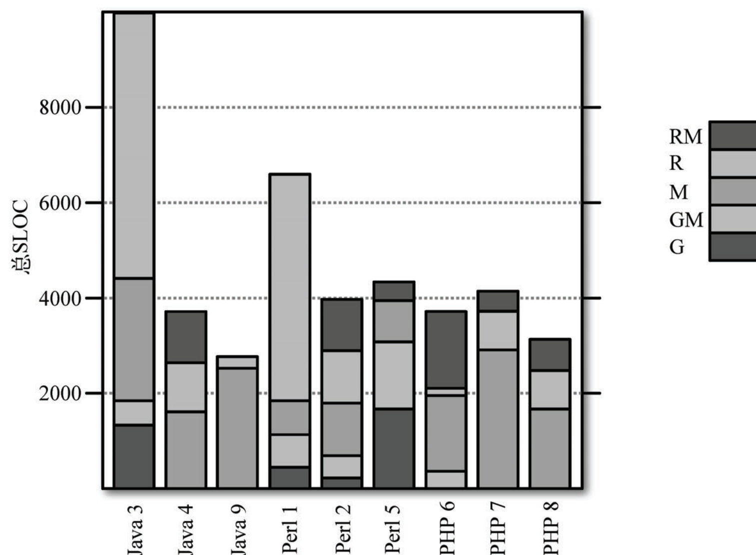


图14-7 工件大小是指每个团队的源代码行数(SLOC)，由文件来源分为：重用然后修改（RM），重用（R），手动编写（M），自动生成然后修改（GM），自动生成（G）。请注意，Java 3组的数据超过80 000（几乎没有重用然后修改的代码），超过其他组10~20倍

这里我们又观察到一些有意思的事情：

- 有些团队修改了重用或者自动生成的代码，有些没有，跟平台关系不大；
- Perl自动生成的代码最多；
- 重用代码的数量差别很大，但看上去跟平台没什么关系；
- 第三组的商用框架最为庞大。

虽然这从学术上看来很有趣，但他们几乎没有多大实际意义。图14-8应该能更好地满足一位务实者的好奇心，这里只考虑了手动编写的文件，把它们的大小与实现的需求做比较。

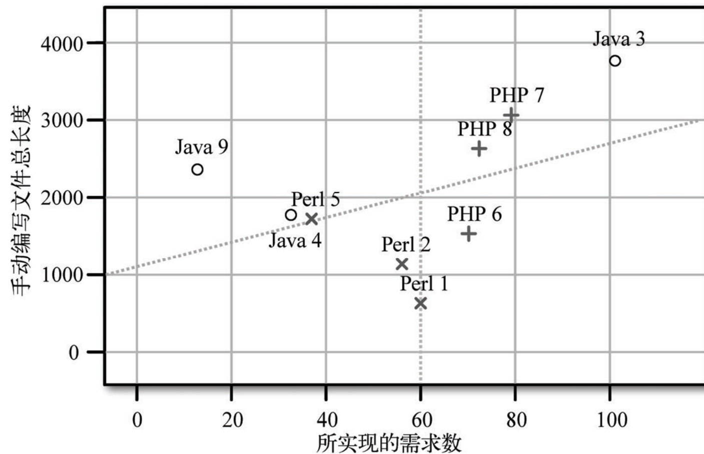


图14-8　基于所实现的需求数（如图14-6所示）的手动编写文件总长度（如图14-7所示）

这个数据表明了我的初始预期至少有一项是正确的：Java项目的大小大于平均值（在趋势线上），Perl的大小小于平均值（在趋势线下）。

14.2.5　比较可修改性

我们关注工件大小的原因一般有两个。首先，它可以用来衡量投入的精力：大型软件往往需要更长的时间来开发。但之前的工作效率数据向我们表明，这个规则在使用的时候要有所保留。其次，工件的大小可以用来衡量可修改性：大工件往往需要更多的精力去修改。在我们的实验中，又是如何的呢？

可修改性由其他两个“性质”所决定：可理解性（我需要多久找出哪里需要改动，如何改动）和模块化性（修改是针对这个产品的一个部分，还是到处都有？）。自20世纪70年代以来，研究人员一直在寻找各种方法来量化这些属性，但迄今为止的所有建议都不很令人满意[2]。因此，在这里我们根据我们所提出的两个简单的修改需求来进行研究。

- 在用户注册时，新增一个文本框表示用户的中间名字的首字母，在用户模型中处理这个数据。用户界面、程序逻辑、用户数据结构和数据库需要什么样的变化？
- 在TTT问卷中添加一个问题及其评估。

对于每一个场景，我们为每个方案都列出了在哪些文件中需要做何种修改。

对于第一种情况，Perl以需要修改的地方最少而胜出，Java和PHP需要作出更多的变化，有些修改并不是那么显而易见。

对于第二种情况，Perl 1，Perl 5，PHP 7，PHP 8都为TTT问卷的文本文件写了一个解释器，所以他们所要做的只是添加一个问题到文件里面去，十分整洁！ Java9组开始也朝着这个方向努力，但没有完成。其他团队的方法都不太直接，比如使用属性文件（Java 3和Java 4组），或源代码中硬编码数组（PHP 6组），或数据库表（Perl 2组）。所有这些都是更难以改变的，需要不止一

个地方的修改。

这些实验带来了很多混合的信息，不过我们认为它们意味着用动态语言的程序员更倾向于用即兴的方法解决问题（相对于标准化）。如果有正确的设计思想（如团队1，5，7，8），可修改性会变得更好。

顺便说一下，国际化很明确不是一个要求，但可以轻松完成，即使使用了解释器的方法。在采用属性文件方法前，Java 3队和Java 4队（以及Perl 1队）曾询问是否允许“客户”在运行时动态修改调查问卷，给他们的答案是“不”。

至于我一开始对于Java有着“干净”的构架设计，而Perl和PHP有着“务实”构架的期望，我个人的结论是，至少在我们要求完成的系统中，我们的研究没有发现 Java具有干净的构架设计（或其他好处）。但Perl的构架的确很务实，也很好用。

14.2.6　比较稳健性和安全性

最后这个部分的结果相对比较壮观（至少在我看来），涉及输入验证、错误检查和安全性的有关问题。对于九个外部（完成需求的数量和表现）和内部（构架设计和技术）都不同的系统来说，很难有一个十分公平的安全性评估。我们最终像可修改性测试中的一样，设计了一些测试场景，并不指望渗透到程序内部，而只希望收集潜在安全漏洞的症状。我们用纯粹的黑盒测试方法，仅在用户界面做些不友好的输入：输入包括HTML标记、SQL语句、汉字、超长字符，等等。图14-9显示了结果。在X轴的项包含：

</…>

　　HTML标记

long

　　十分长的输入

int’l

　　8bit的非西语字符，非8bit的unicode字符

email

　　不正确的邮箱地址

SQL

　　数据和文本输入框中输入SQL语句

cookie

　　执行时，把Cookie关掉

系统的反应被分为：正确OK，可接受（OK），错误（!），安全问题（!!!）。

总结一下我们的发现：

- 基于HTML的CSS

两个PHP的项目和一个Perl项目看上去不太对。Java 4的项目对这个测试没有输出。另两个Java团队的实现正确。

- 插入SQL语句

  所有三个Perl的实现与一个Java实现都会在恶意用户输入的情况下返回SQL异常，并不确定这是否代表了一个实际的漏洞。PHP的实现都正确。
- 长输入

  Perl 2队出错了，其他队伍表现都可接受。
- 国际字符

  Perl 2和Java 3都以“输入过长”拒绝很短的中文输入，PHP的实现都正确。
- 邮箱地址校验

  两个Java的实现与两个Perl的实现都没有地址校验，所有的PHP实现都完成了地址校验。
- 关闭Cookie后的操作

  这令Java 9无声地失败了，而不可思议地令Perl 5也失败了。其他的实现（包括所有的PHP实现），或是操作成功，或是被清晰的出错提示拒绝。

总结一下，我们有两个可能对很多人都很意外的结果。首先，只有PHP 6的实现通过所有测试，或是结果可以接受。其次，除了CSS测试外，其他测试都至少有一个失败的Java实现和一个出错的Perl实现，但没有一个出错的PHP实现。这样看来，至少在精通PHP的队伍手里，PHP比通常想象的要强大。

	</...>	long	int’l.	email	SQL	cookie
PHP 8	!!!	(OK)	(OK)	(OK)	(OK)	(OK)
PHP 7	!!!	(OK)	(OK)	(OK)	(OK)	(OK)
PHP 6	(OK)	(OK)	(OK)	(OK)	(OK)	(OK)
Perl 5	!!!	(OK)	(OK)	!	!	!
Perl 2	(OK)	!	!	(OK)	!	(OK)
Perl 1	(OK)	(OK)	(OK)	!	!	(OK)
Java 9	(OK)	(OK)	(OK)	!	(OK)	!
Java 4				!		(OK)
Java 3	(OK)	(OK)	!	(OK)	!	(OK)

图14-9　黑盒强健性测试和安全性测试的结果。我们评估了对于六种麻烦输入的程序表现。我们只报告看上去有些脆弱的行为，并没有试图真正攻击程序。Java 4缺失一些功能而未能完成这次试验

14.2.7 嘿,“插入你自己的话题”如何

我们在这个实验里也尝试过评估其他一些方面，但有一些太难研究，另一些与平台之间的差异无关。

比如，我们曾预计，在不同平台上团队的开发风格会有所不同。我们试图通过肉眼观察“谁在做什么”的简单分类，每隔十五分钟捕捉一次这些差异（有一个人在九只队伍之间穿梭观察）。我们也在每个团队项目版本库中分析了提交代码信息的“时间、人物、文件”模式。但是，这些数据都没有显示出平台的差异。

当然，我们也试图调查性能和可扩展性。不幸的是，这个项目的功能点太少，不足以进行比较。

同样的，本章中，许多相关的细节都没有明确说明。关于这项研究的详细资料在我2007年的文章[8]中可以找到。稍短但仍然相当精确的是2010年的文章[9]。

14.3 那又怎样

嗯，那么这两个完全不同的研究对我们来说有什么用呢？现有的证据数量太小，无法得出强有力的、可推广的结论。但我们可以结合自己对于软件界的个人理解，使得我们的一些偏见至少比以前看来更有所依据。 如果是我的话，我会得出类似如下的结论。

- 关于编程语言，至少对于文字处理（或类似的普通行为）这样的小型程序，可以保险地说，脚本语言比传统静态类型语言的工作效率更高。
- 至于编码实现的效率，更重要的是避免使用槽糕的程序员，而非错误的语言。有很多方法（和语言）可以帮助你把事情做好。
- 如果在性能上要求非常高或内存使用非常低，C和C++仍然是最优的选择，当且仅当程序员有足够的能力。
- 似乎有种特定的语言文化，有时会导致不同语言的程序员使用不同的方法来设计程序。
- 对Web开发平台而言，当需要开发的Web系统不是特别复杂而团队也能够胜任其开发工作的时候，这个系统所使用的框架就成了编程的语言，而对于此框架的掌握程度远比语言本身的好坏更重要。
- 特别是当有能力的专业人士使用PHP时，PHP比它的名声好很多。
- 各种Web开发平台似乎都有自己的特定文化，有时会导致不同平台的团队使用不同的方法来解决问题。

你的看法可能会有所不同，你将来的路也会有所不同。为了真正了解每个语言和平台的特长，以及使用这些语言或者平台有什么样的特点，我们还有许多诸如此类的研究要做。而且这些问题本身也不停地在变化。

14.4 参考文献

[1] [Boehm 1981] Boehm, B.W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.

[2] [Fenton and Pfleeger 2000] Fenton, N.E., and S.L. Pfleeger. 2000. *Software Metrics: A Rigorous and Practical Approach*. Boston: Thomson.

[3] [Hearty et al. 2008] Hearty, P., N. Fenton, D. Marquez, and M. Neil. 2008. Predicting project velocity in XP using a learning dynamic Bayesian network model. *IEEE Transactions on Software Engineering* 35: 124-137. *http://doi.ieeecomputersociety.org/10.1109/TSE.2008.76*

[4] [Hudak and Jones 1994] Hudak, P., and M.P. Jones. 1994. Haskell vs. Ada vs. C++ vs. awk vs....: An experiment in software prototyping productivity. Yale University, Dept. of CS, New Haven, CT. *http://www.haskell.org/papers/NSWC/jfp.ps*.

[5] [Humphrey 1995] Humphrey, W.S. 1995. A Discipline for Software Engineering. Reading, MA: Addison-Wesley.

[6] [Prechelt 2000a] Prechelt, L. 2000. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Fakultät für Informatik, Universität Karlsruhe, Germany. *http://page.inf.fu-berlin.de/prechelt/Biblio/jccpprtTR.pdf*

[7] [Prechelt 2000b] Prechelt, L. 2000. An empirical comparison of seven programming languages. *IEEE Computer* 33(10): 23-29.

[8] [Prechelt 2007] Prechelt, L. 2007. Plat_Forms 2007: The web development platform comparison—Evaluation and results. Freie Universität Berlin, Institut für Informatik, Germany. *http://www.plat-forms.org/sites/default/files/platformsTR.pdf*

[9] [Prechelt 2010] Prechelt, L. 2010. Plat_forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. Preprint. Forthcoming in *IEEE Transactions on Software Engineering*. *http://doi.ieeecomputersociety.org/10.1109/TSE.2010.22*

[10] [Prechelt and Unger 2001] Prechelt, L., and B. Unger 2001. An experiment measuring the effects of Personal Software Process (PSP) training. *IEEE Transactions on Software Engineering* 27(5): 465-472.

第15章

质量之战：开源软件对战专有软件

Diomidis Spinellis

光说没用，拿代码出来。

——Linus Torvalds[①]

当开发人员比较开源软件与专有软件时，本应该是一场文明辩论，却往往演变成一场充满火药味的战争。其实没有必要如此，因为其中有足够的空间让我们做出一个冷静、客观的比较。

研究人员采用了一些补充方法来研究各种开源软件开发过程的效率。

- 一种方法研究了代码质量，即*内部质量属性*，如注释密度或全局变量的使用情况[41]。
- 另一种方法涉及检测软件的*外部质量属性*，反映软件面对最终用户的情况[22]。
- 然后，除了软件本身，可以观察软件开发过程和检查代码构建和维护的数据指标，比如每周添加多少代码，缺陷解决的速度等[27]。
- 另一些方法讨论了一些特殊的情况，比如Hoepman 和Jacobs[18]通过研究Windows NT系统和Diebold投票系统的一些源代码流出之后是如何遭到攻击的，来研究开源软件的安全性。他们也研究了开源软件如何产生较为清晰的代码，且允许安全检查工具的检测。
- 其他的一些争论基本上就是嘴仗而已，十几年前，Bob Glass [14]就预言了这种趋势的走俏会随着Linux的发展而出现。

虽然多年来很多研究员对于开源软件工件和流程进行了研究（见参考文献[11]、[39]、[8]、[10]、[45]、[3]、[33]、[42]），但开源系统直接与相应的专用软件做比较一直是一个可望而不可及的目标。原因是很难找到与某个开源软件相似并且可比的专用软件，并且说服专用软件的所有者提供源代码来做一个客观的比较。然而，随着Sun的Solaris内核开源（现在甲骨文的一部分）和提供给研究机构的Windows内核的源代码，让我们得以有机会比较一个开源软件的源代码与专有软件的源代码。

在这里，我对代码质量的度量方法基于我收集的四个大型的工业级的操作系统：FreeBSD系统，Linux操作系统，OpenSolaris和Windows研究内核（Windows Research Kernel，简称WRK）。

① Linus Torvalds，Linux创始人。——译者注

本章并不是一个罪案追踪的过程，所以首先说明我的主要发现：这四个系统的代码质量没有非常大的差异。现在，你已经知道了结果，但我仍然建议你阅读下去，因为在下面的部分中，你不仅会发现我是如何得到这个结论的，而且还有很多代码质量指标可以用来评估C编写的软件代码质量，你也可以把它们用在自己的代码上。虽然其中一些指标没有被经验验证过，它们仍基于普遍接受的代码规则，因此代表着对于理想代码属性的大致共识。本人在2008年的国际软件工程[37]会议中，首次报告了这些调查结果，不过这一章包含了更多的细节。

15.1　以往的冲突

历史上写就的每一笔都充满了偏见。

——马克·吐温

二十多年前学者们就开始研究操作系统代码的质量属性[17] [46]。比较开源操作系统的研究[47] [19]，和比较开源与非开源操作系统的研究[41] [27] [31]与我们在此讨论的工作十分相似。

比较Linux和各种伯克利软件套件（BSD）①操作系统的可维护性会发现，Linux操作系统比各种BSD用了更多的、通过全局变量实现的模块间通信（称为共同耦合，common coupling）。我的研究结果中的数据是文件范围内的标识符，不是全局标识符（见图15-11）。此外，对FreeBSD和Linux操作系统的生长动态评价发现两者都是线性增长的，之前认为开源系统比商业系统增长更快是没有根据的[19]。

Paulson和他的同事[27]的研究，比较了三个开源项目（Linux，GCC和Apache）和三个不公开的商业系统的进化模式。他们发现，开源项目的缺陷修正和功能更新速度更快。作为非常受欢迎的项目，这与我们预期的一致。在另一项以代码质量为重点（其内部质量）的研究中[41]，研究人员使用了一个商业工具，用与本实验中类似的指标来评估100个开源应用程序的代码，评估结果的范围从“可接受”到“需要重新写”。然后，他们把结果与该工具供应商提供的商业软件的测试基准作比较发现在软件组织的度量标准下他们评估的模块中只有一半被视为可以接受。同一组[31]的另一个研究评估了开源项目和（半）专有软件的可维护性指数的进化[5]。他们的结论是：所有项目都经历了一段时间的可维护性指数的恶化过程。

15.2　战场

你不能选择你的战场，上天替你选了；但你可以插上你的旗帜，在那没有旗帜飘扬过的地方。

——Nathalia Crane②

图15-1显示了我所研究的操作系统的历史和家谱③。所有四个系统都从1991年～1993年开始独立生存。那个时候，微处理器的价格开始不再难以负担，以此为基础的计算机开始支持32位地

① BSD（Berkeley Software Distribution）是Vnix的衍生系统，20世纪70年代由伯克利加州大学开创。——编者注

② Nathalia Crane，小说家，诗人。——译者注

③ 图15-1的箭头如果你觉得方向指错了，那你是对的。否则重新读一下UML的书籍。

址空间和内存管理，正因为这些原因，现代操作系统如寒武纪生命大爆发一样发展了起来。FreeBSD和OpenSolaris系统，有着共同的祖先，可以追溯到1978年的Unix1BSD版本。FreeBSD是基于BSD/Net2，一个伯克利的Unix源代码发行版，当中清除了AT&T的专有代码。因此，FreeBSD和OpenSolaris都包含在伯克利编写的代码，只是OpenSolaris中包含AT&T的代码。具体来说，OpenSolaris代码起源于1973年的Unix版本，开始时由C语言编写[30]（参考文献第54页）。2005年，Sun在开源许可证下发布了大部分Solaris的源代码。

Linux从无到有，都是为了在Tanenbaum①的面向教学、兼容POSIX的Minix操作系统的基础上[43]，构建一个功能更丰富的版本。虽然Linux借用了Minix和Unix的想法，但它并没有用它们的代码[44]。

Windows NT的版权历史可以追溯到DEC的VMS系统，这两个项目的首席工程师都是David Cutler。Windows NT是微软作为对Unix的回应而开发的，最初作为IBM OS / 2替代系统，后来更换了16位Windows代码库。本研究中的Windows研究代码（WRK），包括了64位Windows内核，是微软提供用于研究的[28]。该内核是用C以及一些小扩展编写的。设备驱动程序、即插即用支持、电源管理、虚拟DOS子系统等未包含在内。这些未被包含的部分解释了WRK和其他三个内核巨大的规模差异。

虽然我研究的所有四个系统都以源代码方式提供，但其开发方法明显不同。微软和Sun的工程师在其公司内构建Windows NT和 Solaris，如果有外部参与的话，也是很小的。（OpenSolaris作为开源项目，时间很短，因此，在我研究的大致印象中，只有很少的代码可能是由Sun外部的开发人员开发的。）此外，Solaris是严格按照标准流程开发的[7]，而Windows NT则采用较为轻量级的开发方法[6]（文献资料223，263，273～274页）。FreeBSD和Linux都使用开源的开发方法[9]，但其发展过程也是不一样的。FreeBSD主要是由约220个提交者组成的一个没有等级划分的小组开发的，他们访问共享的软件库，这个软件库最初基于CVS，目前基于Subversion[20]。相比之下，Linux的开发者分为四层，类似一个金字塔。在底部的两个层次有成千上万的开发者开发补丁提供给约560名子系统维护人员。在金字塔的顶端，Linus Torvalds，由被信任的副手组协助，是唯一负责将补丁加入Linux树的人[29]。如今，Linux开发人员用git协调代码变化，git是一个为此目的而建的分布式版本控制系统。

我报告里的指标多数是通过执行SQL查询来的，SQL语句查询了一个关系型数据库②，该数据库包含了每个系统代码的组成部分：模块、标识符、令牌、方法、文件、注释和它们之间的关系。图15-2显示了数据库的架构。我通过运行C语言的CScout重构浏览器为每个系统建立数据库（见参考文献[35]、[38]），并且每个系统都运行在几个特殊的处理器配置上（处理器的具体配置包括一些特有的宏定义和文件，因此会以不同的方式处理代码）。为了完整地处理代码，CScout必须定义一个配置文件，定义处理每个编译单元（C文件）的环境。对于FreeBSD和Linux的内核，我检查了GNU C编译器、连接器以及一些shell命令之间的调用关联。从中记录下其调用时的具体参数格式（主要包括文件路径和宏定义等），然后可以用来构造出CScout的配置文件。对于

① Andrew S. Tanenbaum领导编写的MINIX，是一个用于操作系统教学的类UNIX小型操作系统。——译者注

② 这个数据库有1.4亿多条记录，查看相应的查询语句请访问http://www.spinellis.gr/sw/4kernel/。

OpenSolaris和WRK，我只是对我调查的所有配置做了一个编译连接（build），记录了在日志文件中出现的执行过的命令，然后通过日志中的这些命令来处理编译和链接。

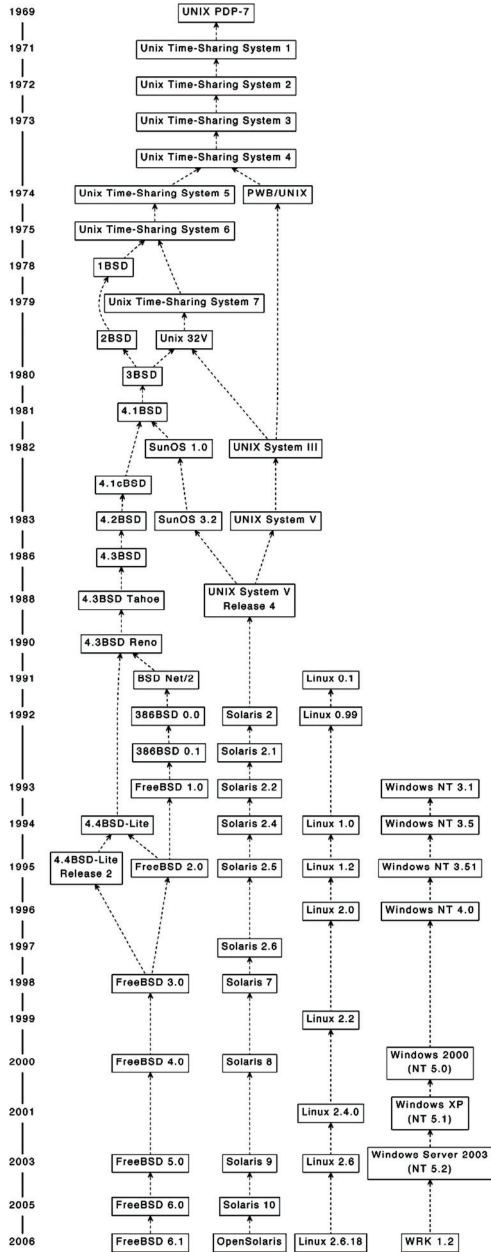


图15-1　四个系统的历史和发展体系

图15-2　代码分析的数据库结构

为了减少因为选择度量指标引入的偏差，我在建立测量这些指标的机制之前就选择并确定了度量指标。这可以帮助我避免基于结果而选择指标的偏差。但是，这种事前的选择会产生很多度量指标，比如说每行字符数，诸如这样的指标并不提供任何有意义的信息，当然也不会得出一个明确的赢家或输家。另一方面，我对指标的选择并不是完全盲目的，因为当我设计实验时，我已经十分熟悉FreeBSD的内核源代码，也得到了Linux、第9个研究版Unix和一些Windows设备驱动程序的源代码。

这项研究的其他局限性包括：参与研究的系统数量太少（当然是指那些大型而且重要的系统）；所采用度量指标的语言很具体；只覆盖了软件质量指标中的可维护性和可移植性。最后一项限制也意味着，这项研究没有考虑在运行时决定的一些重要的软件质量特性：功能性、可靠性、可用性和效率。不过，这些缺少的属性往往取决于一些非开发因素：配置，优化和工作负荷。研究它们将引入更多的主观偏见，比如对于某些操作系统或者工作负荷来说不适当的配置。比较操作系统之间的安全性或性能的研究所引起的争议恰恰证明了这些方法的不易。

WRK源代码与其他操作系统间巨大的规模差异并没有像当初认为的那样会有很大问题。一项关于FreeBSD多个组件的可维护性指数[5]分布的早期研究表明其可维护性指数也是均匀分布，在两端[36]（图7-3）有极个别例外。这也就是说，我们可以通过研究一个大型软件系统的一个部分来得出对该系统的评价。因此，WRK中的代码，可以被视为一个子集，代表完整的Windows操作系统内核。

15.3　开战

没有任何作战计划在与敌人交锋后还有效。

——老赫尔穆特·冯·毛奇①

① Helmuth von Moltke the Elder，德意志名将，参谋长，陆军元帅。——译者注

表15-1显示了被研究系统的关键属性。质量指标大致可分为文件组织、代码结构、代码风格、预处理和数据组织。当可以用一个数字表示某个度量指标时，我在表格中列出4个系统的值，并在左边说明了理想情况下，该数字应该是高（↑），或是低（↓），或趋近某个特定值（例如，≅ 1）。在其他情况下，我们必须着眼于各个值的分布，为此我使用烛台图，如图15-3所示。图中的每个元素表示6个值：

- 最小值，在该线的底部；
- 低（25%）四分位值，在盒的底部；
- 中位数（数值用来分离较高的一半和较低的一半），盒子的水平线；
- 高（75%）四分位值，在盒的顶部；
- 最大值，在该线的顶部；
- 算术平均值，菱形。

超出图示范围的最大最小值用虚线表示，并有数字标示它们的准确值。

表15-1　四个操作系统的主要指标

指　　标	FreeBSD	Linux	Solaris	WRK
版本	HEAD 2006-09-18	2.6.18.8-0.5	2007-08-28	1.2
配置	i386 AMD64 SPARC64	AMD64	Sun4V、Sun4V、SPARC	i386AMD64
代码行数（千）	2599	4150	3000	829
注释（千）	232	377	299	190
声明（千）	948	1772	1042	192
源文件	4479	8372	3851	653
链接模块	1224	1563	561	3
C方法	38 371	86 245	39 966	4820
宏定义	727 410	703 940	136 953	31 908

15.3.1　文件组织

在C语言中，源代码文件在构建一个系统中发挥了重要作用。一个文件圈定了范围边界，而它所在的目录决定了搜索所包含头文件的路径[16]。因此，适当地组织源代码中的声明、定义和文件所处目录，决定了该系统的模块化[26]。

图15-3显示了C源代码文件和头文件的长度。大多数文件都小于2000行代码。过长的文件（如在OpenSolaris和WRK中的C文件）往往都有问题，因为它们难以管理，有很多依赖，并且违反了模块化的宗旨。事实上，最长的头文件有27000行（WRK中的Winerror.h文件），集合了来自30个不同地方的错误信息，其中大部分与Windows内核无关。

另一个相关方法计算了源代码中定义的实体数而不是行数。在C代码中，它计算全局函数的数量。在头文件中，一个重要的实体是一个结构，在C语言中最接近类的抽象。图15-4显示了每个C文件中声明的数量和每个头文件中的聚集数（结构或集合）。理想情况下，这两个数字应该

较小，表明适当的独立性。OpenSolaris和WRK中的C文件不如其他系统，同时有相当一部分WRK中的头文件看起也不怎样，因为它们每个都定义了10多个结构。

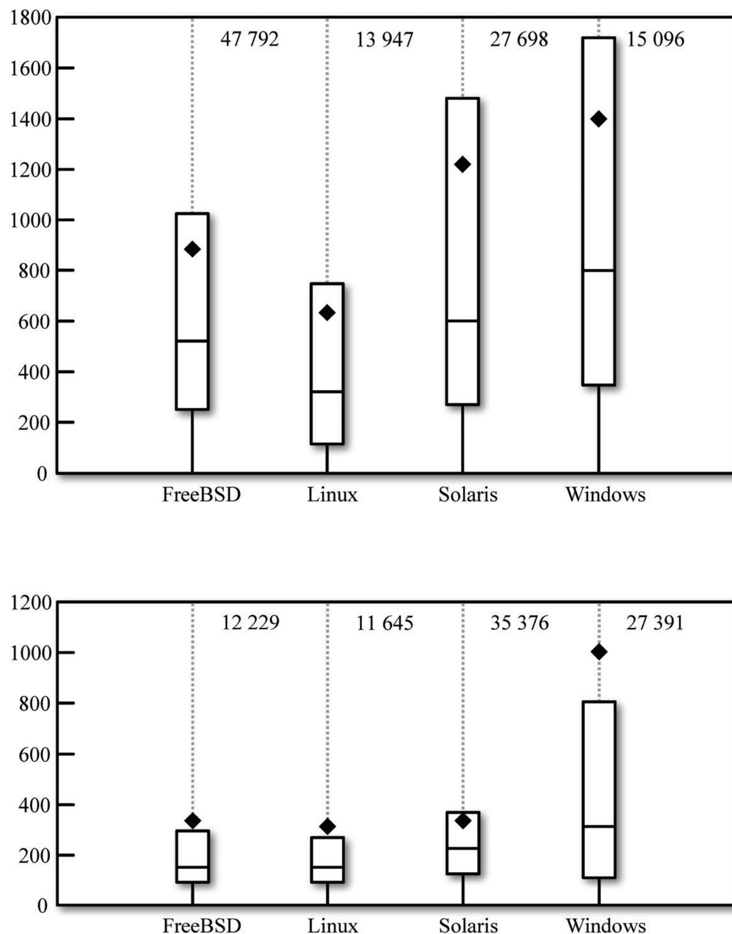


图15-3 源代码（上）和头文件（下）行数

我所研究的四个系统有着很有趣的目录结构。正如图15-5到图15-8中显示的数字，四个系统中的三个有着相似的扁平结构。WRK的小体积和复杂性反映了微软已经将Windows内核的大部分从中剥离了出去。我们看到，Linux的目录在整个源代码树中分布得相对均匀，而在FreeBSD和OpenSolaris中，某些目录是簇拥在一起的。这可能是长时间逐步发展的结果，毕竟这两个系统背后都有20多年的历史（图15-1）。Linux目录的均匀分布也反映了开发的分散性。

在一个更高层次的粒度上，我考察了一个目录中的文件数。同样地，在一个目录下放很多文件，就好像一个模块中有许多内容。大量文件会让开发人员迷惑，开发人员用例如grep的工具来搜索文件群，也会导致共享头文件的标识符冲突。表15-2中是研究得出的数据，Linux明显落后。

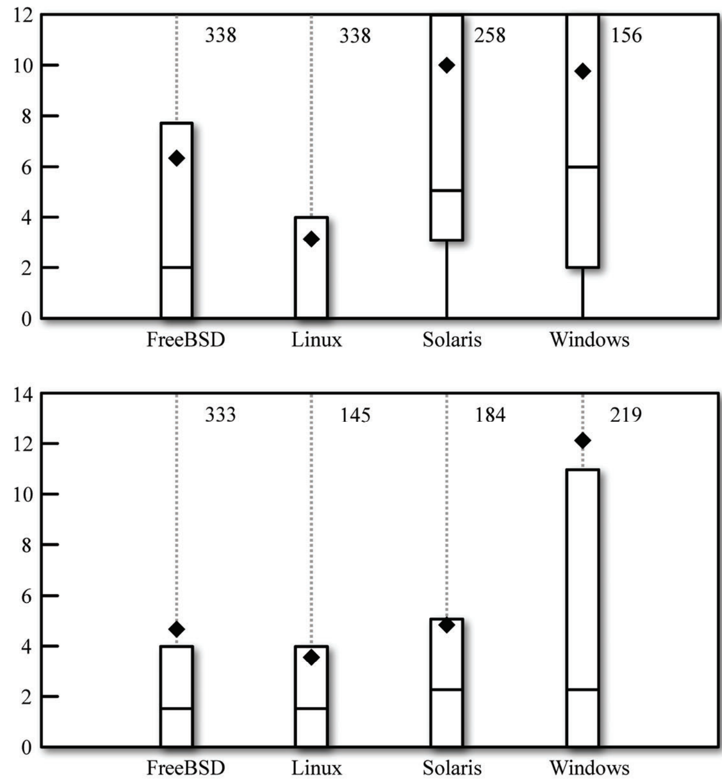


图15-4　全局函数（上）和结构（下）

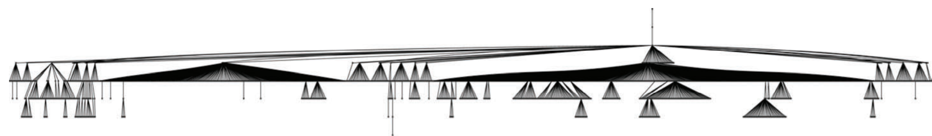


图15-5　FreeBSD的目录结构



图15-6　Linux的目录结构

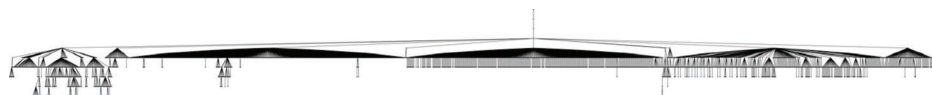


图15-7　OpenSolaris的目录结构

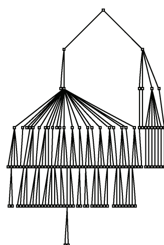


图15-8　Windows研究内核的目录结构

表15-2　文件结构统计

指　　标	理　想　值	FreeBSD	Linux	Solaris	WRK
每个目录的文件数	↓	6.8	20.4	8.9	15.9
C源文件的头文件数	≅ 1	1.05	1.96	1.09	1.92
文件的平均结构复杂度	↓	2.2×10^{14}	1.3×10^{13}	5.4×10^{12}	2.6×10^{13}

表格中的第二行表示C源文件数与头文件数的比率。我用以下的SQL语言来计算：

```
select (select count(*) from FILES where name like '%.c') /
 (select count(*) from FILES where name like '%.h')
```

在标准的C代码风格指南中，要求每一个模块的接口定义在单独的头文件中，而在相应的C文件中实现。因此，C文件与头文件的最优比率是1∶1，差异很大可能意味着接口和实现之间的区别模糊。对于一个小系统这也许是可以接受的（在awk编程语言的实现里面，该比率是3 /11），但对于一个有成千上万文件的系统来说，这就是个问题。在这项指标中，所有系统的评分都是可接受的。

表15-2的最后一行代表文件间关系的复杂性。我用有向图以及图中的节点（表示一个文件）来研究这个问题。文件引用外部其他文件中的定义或声明元素的数量，称为引用数（fanout）。例如，一个C文件，使用了包括`FILE`，`putc`，和`malloc`（定义在stdlib.h和stdio.h中）的三个符号，则该C文件的引用数为3。相应地，我定义了一个文件的被引用数（fanin）。因此，在前面的例子里，stdio.h的被引用数是2。我用Henry和Kafura的信息流度量法[17]来看文件之间的对应关系。

表格中数据的计算公式是：

$$(\text{被引用数} \times \text{引用数})^2$$

我用CScout数据库中`INCTRIGGERS`表的数据进行计算，该表存储每个文件中的链接到其他文件的符号（symbol）。

```
select avg(pow(fanout.c * fanin.c, 2)) from
  (select basefileid fid, count(definerid) c from
    (select distinct BASEFILEID, DEFINERID, FOFFSET from INCTRIGGERS) i2
    group by basefileid) fanout
  inner join
  (select definerid fid, count(basefileid) c from
    (select distinct BASEFILEID, DEFINERID, FOFFSET from INCTRIGGERS) i2
    group by definerid) fanin
  on fanout.fid = fanin.fid
```

计算方式如下：最内层的SELECT语句从各个文件相关的外部定义和引用表中找到其中互不重复的标识符集合，及其定义或引用所在的文件。然后，中层的select语句计算每个文件中的标识符数量，最外面的select语句把各个文件中的外部定义以及引用相互联接，从而计算得到相应的信息流指标。这个值很大的话，意味着频繁的变动或是结构缺陷。

15.3.2　代码结构

这四个系统的代码结构说明了如何通过不同的控制结构和不同的侧重点解决相似的问题，也使我们能够一窥每个系统设计。

图15-9显示了跨方法的扩展圈复杂度（Cyclomatic complexity）指标[24]。它衡量了每个方法中所包含的独立路径的数量。这个数字包括布尔和条件语句（因为它们引入额外的路径），但不包括多向switch语句，因为它会不均衡地影响相似代码的研究结果。度量的目的是要衡量一个程序的可测试性、可理解性以及可维护性[13]。在这方面，Linux的成绩优于其他系统，WRK最糟。这些数字也显示了每个方法中C语句的数目。理想的情况下，这应该是一个小数目（例如，约20），函数的所有代码都可以显示在屏幕上。这点上，Linux再次优于其他系统。

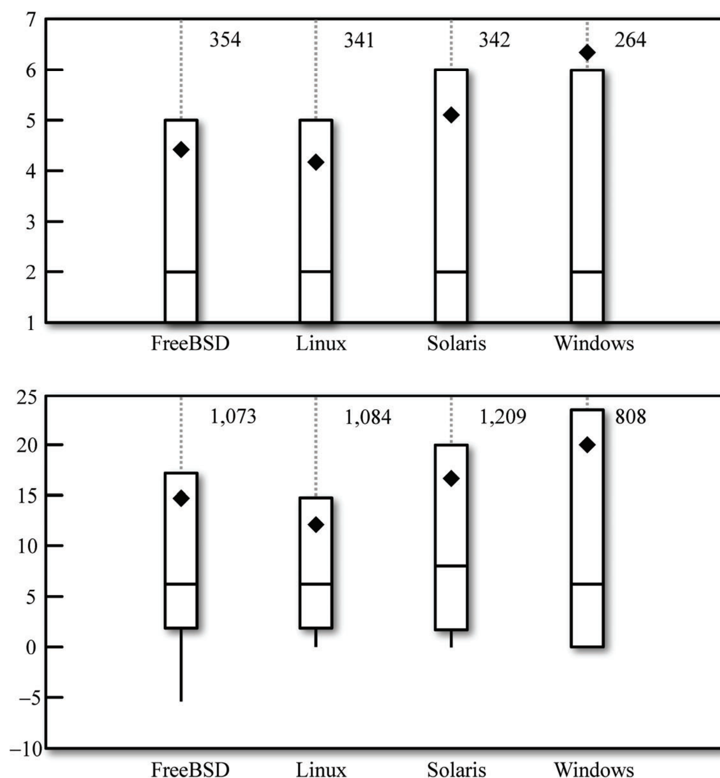


图15-9　圈复杂度分布（上），每个方法中C语句数目（下）

图15-10显示了Halstead量复杂度[15]。对于给定代码来说，需要考察代码中4个数值：

- $n1$

  不同操作符的数量。
- $n2$

  不同操作数的数量。
- $N1$

  所有操作符的数量。
- $N2$

  所有操作数的数量。

利用这4个数值，计算量复杂度的公式为：

$(N1+N2)\times\log_2(n1+n2)$

比如，对于这个表达式：

```
op = &(!x ? (!y ? upleft : (y == bottom ? lowleft : left)) :
(x == last ? (!y ? upright : (y == bottom ? lowright : right)) :
(!y ? upper : (y == bottom ? lower : normal))))[w->orientation];
```

这4个变量有如下取值：

- $n1$

  ```
  = & () ! ?: == [] -> (8)
  ```
- $n2$

```
    bottom last left lower lowleft lowright normal op orientation right upleft upper
upright w x y  (16)
```

- $N1$

  27
- $N2$

  24

Halstead量复杂性背后的理论说明，该值如果低的话则表示代码好理解。但是，这项度量指标本身经常受到批评。在这里，Linux的成绩最好，WRK比其他系统都差。

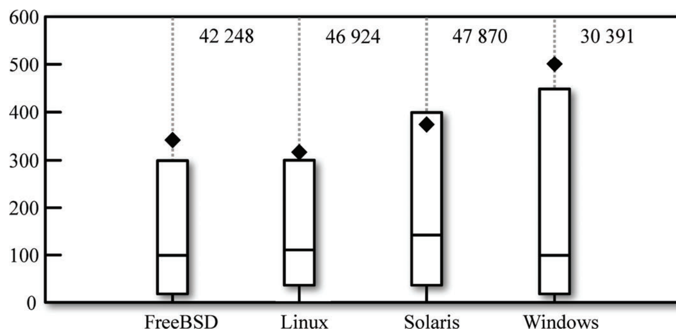


图15-10 函数的Halstead复杂度

退一步看函数之间的交互，图15-11反映了功能耦合度，它显示了函数内一个标识符来自于编译单元作用域（文件作用域的标识符或静态标识符）或是项目作用域（全局对象）的百分比。这两种形式的耦合都是不可取的，全局标识符比文件作用域标识符更为糟糕。全局范围中，Linux的共同耦合情况比其他系统好，但（可能因为如此）文件作用域的分数比其他系统更差。其他系统比较均衡。

表15-3是与代码结构有关的其他指标。全局函数指整个系统都可见的函数。这种函数的个数在WRK中（接近100%，手工验证过）高得惊人。然而，这可能反映了微软使用不同的技术，比如连接到共享库（DLL）和显式导出符号（Symbol），以避免标识符冲突。

表15-3　代码结构指标

指　　标	理　想　值	FreeBSD	Linux	Solaris	WRK
全局函数百分比	↓	36.7	21.2	45.9	99.8
结构严格的函数百分比	↑	27.1	68.4	65.8	72.1
标记语句百分比	↓	0.64	0.93	0.44	0.28
函数参数平均数量	↓	2.08	1.97	2.20	2.13
平均最大嵌套深度	↓	0.86	0.88	1.06	1.16
每个语句的记号数	↓	9.14	9.07	9.19	8.44
重复代码的记号数百分比	↓	4.68	4.60	3.00	3.81
函数的平均结构复杂度	↓	7.1×10^4	1.3×10^8	3.0×10^6	6.6×10^5

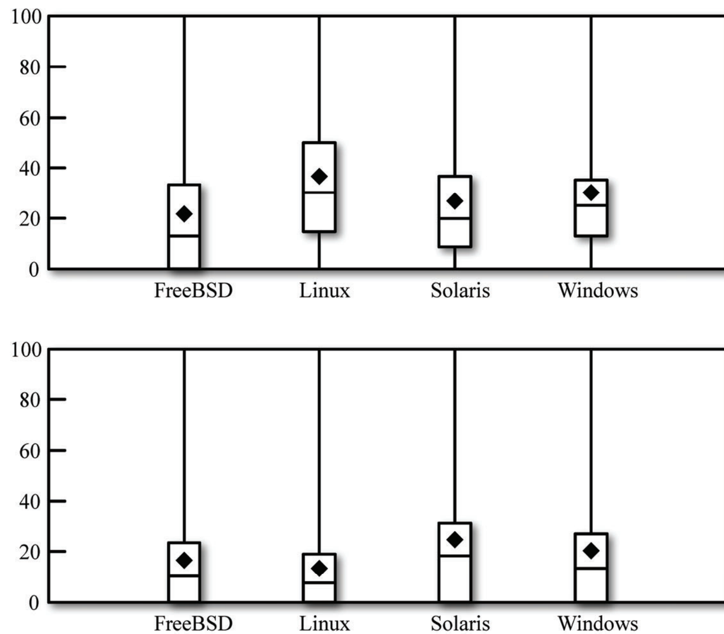


图15-11　文件级（上）和全局（下）的一般耦合

结构严格的函数符合如下规定：单点结束，没有goto语句。这点很容易被验证。通过以下语句来观察每个函数中的关键词就可以计算这些语句的百分比：

```
select 100 -
(select count(*) from FUNCTIONMETRICS where nreturn > 1 or ngoto > 0) /
(select count(*) from FUNCTIONMETRICS) * 100
```

沿着同样的思路，标记语句的百分比表明了goto的目标：这违反了结构编程原则。我这里计算标记语句而不是goto语句，因为众多分支目标比众多分支源头更让人困惑。通常会有多个goto语句指向同一个标记语句，用来退出函数去做一些清理工作，这与异常处理的finally语句相似。

函数参数的数量体现了接口的质量。当一个函数有很多参数时，把参数包装成一个结构会减低混乱，也可以易于优化，提高性能。

最深嵌套的平均深度与每个语句的标记数表示了代码的可理解性。这基于深度嵌套和长语句不易于理解的理论[2]。

重复代码意味着缺陷[23]和维护问题[36]（参考文献的413~416页）。相应的度量指标（重复代码记号数百分比）表示至少在一个克隆代码中出现过的代码段落百分比。我用CCFinderX①的工具寻找重复代码，用Perl脚本（见示例15-1）来找出相应的比率。

示例15-1　从CCFinderX来找出重复代码的比率

```
# Process CCFinderX results
open(IN, "ccfx.exe P $ARGV[0].ccfxd|") || die;
while (<IN>) {
    chop;
    if (/^source_files/ .. /^\}/) {
        # Process file definition lines like the following:
        # 611     /src/sys/amd64/pci/pci_bus.c        1041
        ($id, $name, $tok) = split;
        $file[$id][$tok - 1] = 0 if ($tok > 0);
        $nfile++;
    } elseif (/^clone_pairs/ .. /^\}/) {
        # Process pair identification lines like the following for files 14 and 363:
        # 1908    14.1753-1832    363.1909-1988
        ($id, $c1, $c2) = split;
        mapfile($c1);
        mapfile($c2);
    }
}

# Add up and report tokens and cloned tokens
for ($fid = 0; $fid <= $#file; $fid++) {
    for ($tokid = 0; $tokid <= $#{$file[$fid]}; $tokid++) {
        $ntok++;
        $nclone += $file[$fid][$tokid];
    }
```

① http://www.ccfinder.net/。

```
}
print "$ARGV[0] nfiles=$nfile ntok=$ntok nclone=$nclone ", $nclone / $ntok * 100, "\n";

# Set the file's cloned lines to 1
sub mapfile
{
    my($clone) = @_;
    my ($fid, $start, $end) = ($clone =~ m/^(\d+)\.(\d+)\-(\d+)$/);
    for ($i = $start; $i <= $end; $i++) {
        $file[$id][$i] = 1;
    }
}
```

最后，函数的平均结构复杂度还是用Henry和Kafura的信息流度量法[17]来找出函数间的联系。理想中，这个值要尽量地小，意味着函数提供者和使用者被适当地分离开。

15.3.3　代码风格

缩进、间距、标识符命名、常量名称、命名规则的各种选择都会在功能不变的情况下影响代码的表达（参见参考文献[21]、[12]、[1]、[40]）。在大多数正常情况下，代码风格的一致性比选择哪种代码风格更重要。

在这项研究中，我用indent程序来排版每个系统的所有代码，统计有多少行代码被indent修改了，以此来衡量各系统代码风格的一致性。表15-4的第一行就是这一项实验的结果。可以修改Indent的参数来符合特定的排版要求。比如你可以修改缩进量与括号的位置。为了确定每个系统的排版样式（format style）和使用合适的排版选项，首先，我依次尝试设定15种不同的数值型标记，以及逐个开启或关闭 55个布尔型标记来找到合适的标记组合（参见示例15-2和示例15-3shell脚本）。继而我以此来选择能产生与原有代码风格最一致的一组标记组合。例如，对OpenSolaris的源代码，用indent及其默认参数将重新排版74%的代码行。一旦确定了相应的标记（-i8 -bli0 -cbi0 -ci4 -ip0 -bad -bbb -br -brs -ce -nbbo -ncs -nlp -npcs），这个数值将减少至16%。

表15-4　代码风格指标

指　　标	理　想　值	FreeBSD	Linux	Solaris	WRK
代码行风格一致百分比	↑	77.27	77.96	84.32	33.30
typedef风格一致百分比	↑	57.1	59.2	86.9	100.0
聚合标记风格一致百分比	↑	0.0	0.0	20.7	98.2
每行的字数	↓	30.8	29.4	27.2	28.6
操作数数值常数百分比	↓	10.6	13.3	7.7	7.7
不安全的类函数宏命令百分比	↓	3.99	4.44	9.79	4.04
注释拼写错误百分比	↓	33.0	31.5	46.4	10.1
特殊注释拼写错误百分比	↓	6.33	6.16	5.76	3.23

示例15-2 确定系统缩进格式选项

```
DIR=$1
NFILES=0
RNFILES=0

# Determine the files that are OK for indent
for f in `find $DIR -name '*.c'`
do
    # The error code is not always correct, so we have to grep for errors
    if indent -st $f 2>&1 >/dev/null | grep -q Error:
    then
        REJECTED="$REJECTED $f"
        RNFILES=`expr $RNFILES + 1`
        echo -n "Rejecting $f - number of lines: "
        wc -l <$f
    else
        FILES="$FILES $f"
        NFILES=`expr $NFILES + 1`
    fi
done

LINES=`echo $FILES | xargs cat | wc -l`
RLINES=`echo $REJECTED | xargs cat | wc -l`

# Format the files with the specified options
# Return the number of mismatched lines
try()
{
    for f in $FILES
    do
        indent -st $IOPT $1 $f |
        diff $f -
    done |
    grep '^<' |
    wc -l
}

# Report the results in a format suitable for further processing
status()
{
    echo "$IOPT: $VIOLATIONS violations in $LINES lines of $NFILES files \
($RLINES of $RNFILES files not processed)"
}

# Determine base case
VIOLATIONS=`try`
Status
```

示例15-3　确定系统缩进格式选项（续）

```
# Try various numerical options with values 0-8
for try_opt in i ts bli c cbi cd ci cli cp d di ip l lc pi
do
    BEST=$VIOLATIONS
    for n in 0 1 2 3 4 5 6 7 8
    do
        NEW=`try -$try_opt$n`
        if [ $NEW -lt $BEST ]
        then
            BNUM=$n
            BEST=$NEW
        fi
    done
    if [ $BEST -lt $VIOLATIONS ]
    then
        IOPT="$IOPT -$try_opt$BNUM"
        VIOLATIONS=$BEST
        status
    fi
done

# Try the various Boolean options
for try_opt in bad bap bbb bbo bc bl bls br brs bs cdb cdw ce cs bfda \
    bfde fc1 fca hnl lp lps nbad nbap nbbo nbc nbfda ncdb ncdw nce \
    ncs nfc1 nfca nhnl nip nlp npcs nprs npsl nsaf nsai nsaw nsc nsob \
    nss nut pcs prs psl saf sai saw sc sob ss ut
do
    NEW=`try -$try_opt`
    if [ $NEW -lt $VIOLATIONS ]
    then
        IOPT="$IOPT -$try_opt"
        VIOLATIONS=$NEW
    fi
    status
done
```

图15-12描述了两类重要的C标识符的长度分布：全局可见的对象（变量和函数）和用来鉴别聚合的标签（结构和联合）。因为每个类通常使用独立的命名空间，选择不同的可识别名称很重要（见参考文献[25]第31章）。对于这些标识符类，较长的名称是比较可取的，这两种情况下，WRK最棒，任何使用Windows API编程的程序员应该很容易猜到这一点。

表15-4中相关代码风格的其他一些指标出现。为了测量的一致性，我通过代码检测来确定`typedef`和聚合标签的命名规则，然后计算不匹配规则的标识符。这里我运行了两个SQL查询，一个在类Unix系统，一个在WRK中：

```
select 100 * (select count(*) from IDS where typedef and name like '%_t') /
  (select count(*) from IDS where typedef)
select 100 * (select count(*) from IDS where typedef and name regexp '^[A-Z0-9_]*$') /
  (select count(*) from IDS where typedef)
```

以下是检查编程风格是否违反编程准则的其他三个指标：

- 过长的代码行（每行字符数）；

- ❑ 代码中直接使用的“魔法”数字（操作数中出现数值常量的百分比）；
- ❑ 置于If语句之后会导致异常行为的类函数宏[①]的定义（不安全的函数宏的百分比）。

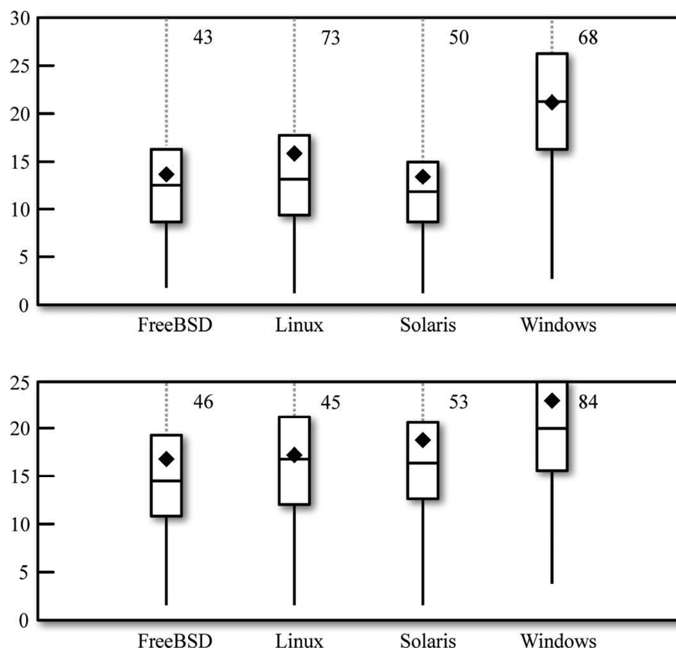


图15-12 全局标识符长度（上），聚合标识符长度（下）

下面的SQL查询粗略计算了不安全的类函数宏的数量，寻找那些含有多个语句，但没有do关键字的宏。

结果显示的是一个下界，因为查询可能会遗漏其他不安全的宏，比如那些含有if语句的不安全宏：

```
select 100.0 * (select count(*) from FUNCTIONMETRICS left join
  FUNCTIONS on functionid  = id where defined and ismacro and ndo = 0 and nstmt > 1) /
(select count(*) from FUNCTIONS where defined and ismacro)
```

注释是代码风格的另一个重要因素。但很难客观判断代码注释的质量。注释可能是多余的甚至是错误的。我们不能编程来判断注释质量，但是我们可以很容易地测量注释密度。图15-13显示了C文件中，注释字符对语句的比率。在头文件中，我计算了定义元素对注释数量的比率，通常定义元素需要注释（枚举、聚合及其成员、变量声明和类函数的宏）。在这两种情况下，都排除了内容很少的文件。WRK具有明显的一致性，Linux在这方面表现较差。有趣的是，注释平均值要比中位数高，表明一些文件中的注释比其他文件多许多。

```
select nccomment / nstatement from FILES where name like '%.c' and nstatement > 0
```

```
select (nlcomment + nbcomment) / (naggregate + namember + nppfmacro + nppomacro +
  nenum + npfunction + nffunction + npvar + nfvar) from FILES
```

① 类函数宏包含了不止一个语句，并且代码体在do...while(0)中，使它们像一个对于真实函数的调用。

```
where name like '%.h' and naggregate + namember + nppfmacro + nppomacro > 0
  and nuline / nline < .2
```

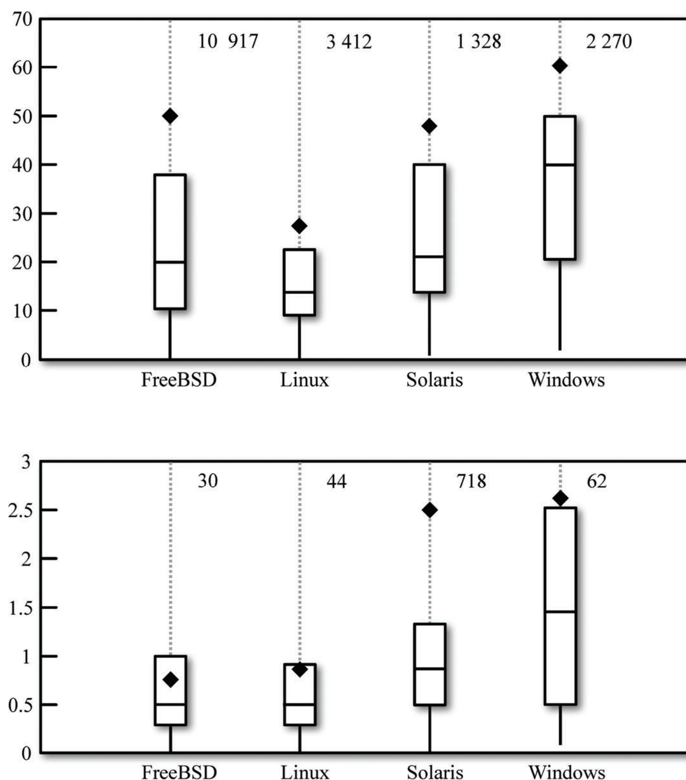


图15-13　C文件中的注释密度（上），头文件中的注释密度（下）

同时，我也计算了拼写错误，这也是代码质量的指标之一。我用所有系统的标识符和文件名组成了一个字典，基于该字典用aspell来检查注释（参见示例15-4中的脚本）。WRK的错误很少，根据相应的文档，表明代码发布之前做过拼写校验。

示例15-4　在Cscout的数据库中计算注释以及拼写错误

```
# Create personal dictionary of correct words
# from identifier names appearing in the code
PERS=$1.en.pws
(
echo personal_ws-1.1 en 0
    (
    mysql -e 'select name from IDS union select name from FUNCTIONS union
        select name from FILES' $1 |
    tr /._ \\n |
    sed 's/\([a-z]\)\([A-Z]\)/\1\
\2/g'
    mysql -e 'select name from IDS union select name from FUNCTIONS union
        select name from FILES' $1 |
```

```
        tr /. \\n
        ) |
    sort -u
    ) >$PERS
    # Get comments from source code files and spell check them
    mysql -e 'select comment from COMMENTS left join FILES
        on COMMENTS.FID = FILES.FID where not name like "%.cs"' $1 |
    sed 's/\\[ntrb]//g' |
    tee $1.comments |
    aspell --lang=en --personal=$PERS -C --ignore=3 --ignore-case=true  \
        --run-together-limit=10 list >$1.err
    wc -w $1.comments   # Number of words
    wc -l $1.err        # Number of errors
```

虽然我没有客观衡量可移植性，但用CScout处理源代码的过程让我感觉到各系统的源代码对于不同的编译器可移植性会不同。Linux和WRK中的代码似乎是与一个特定的编译器紧密绑定的。Linux使用GNU C编译器提供的各种语言的扩展，有时还包括伪装成被gcc认作C语法的汇编代码（见示例15-5）。WRK中使用相当少的语言扩展，但依赖于微软编译器提供的C语言的`try catch`扩展。FreeBSD只用了少数gcc的扩展，并用宏标出。OpenSolaris表现不俗，是源码中唯一不需要Cscount之外的扩展来编译的。

示例15-5　Linux内核中memmove的定义

```
void *memmove(void *dest, const void *src, size_t n)
{
        int d0, d1, d2;

        if (dest < src) {
                memcpy(dest,src,n);
        } else {
                __asm__ __volatile__(
                        "std\n\t"
                        "rep\n\t"
                        "movsb\n\t"
                        "cld"
                        : "=&c" (d0), "=&S" (d1), "=&D" (d2)
                        :"0" (n),
                         "1" (n-1+(const char *)src),
                         "2" (n-1+(char *)dest)
                        :"memory");
        }
        return dest;
}
```

15.3.4　预处理

C语言的语言规范与它的（不可或缺的）预处理器之间的关系不是一句话就能说清楚的。尽管C和真正的程序很大程度上依赖于预处理器，但它的特性常常造成可移植性，可维护性和可靠性的问题。预处理器，作为一个强大的，但粗糙的手段，在识别标识符的作用范围上很弱，解析和重构没有预处理过的代码的能力也不强，在不同的平台编译代码的方式也不好。因此，大多数

C编程指南推荐适度地使用预处理器结构（preprocessor construct）。也因为这个原因，以C为基础的现代语言都试图提供可以取代C预处理器的其他规范选择。例如，C++提供的常量和强大的模板替代C的宏，C#提供类预处理功能用来帮助条件编译和代码生成。

预处理器的功能可通过预处理器运行时代码的膨胀或收缩量来衡量。图15-14包含两个这样的衡量手段：函数体（即代码膨胀），函数体外的元素（即数据定义和声明）。这两个测量了计算进入预处理器的标记与预处理结束时的标记的比率。下面是本人计算函数体内代码膨胀用的SQL查询语句：

```
select nctoken / npptoken from FUNCTIONS
inner join FUNCTIONMETRICS on id = functionid
where defined and not ismacro and npptoken > 0
```

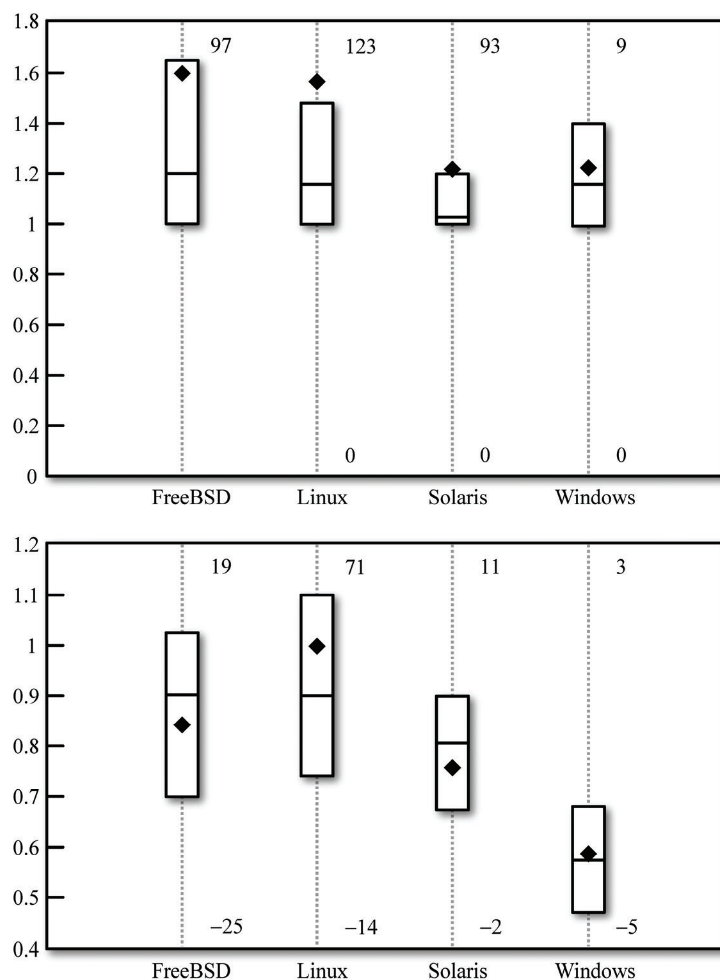


图15-14　函数内的预处理膨胀（上），文件内的预处理膨胀（下）

膨胀和收缩都令人担忧。膨胀象征着复杂的宏，收缩则表示条件编译，这都是有害的[34]。因此，这些指标的值应该徘徊在1。在关于函数的图中，OpenSolaris的成绩优于其他系统，FreeBSD最糟。在关于文件的情况下，WRK比其他系统都差许多。

表15-5显示了其余4个衡量预处理器不安全因素的指标。

- 头文件中的指令（directives）（多数情况下需要）。
- 头文件中非-`#include`的指令（很少用到）。
- 函数中的预处理器命令（会产生不确定值）。
- 函数中的预处理条件（影响可移植性）。

表15-5 预处理指标

指标	理想值	FreeBSD	Linux	Solaris	WRK
头文件中的指令百分比	↓	22.4	21.9	21.6	10.8
头文件中非`#include`的指令百分比	↓	2.2	1.9	1.2	1.7
函数中的预处理器命令百分比	↓	1.56	0.85	0.75	1.07
函数中的预处理条件百分比	↓	0.68	0.38	0.34	0.48
定义函数体内类函数的宏百分比	↓	26	20	25	64
单独标识符中的宏百分比	↓	66	50	24	25
标识符中的宏百分比	↓	32.5	26.7	22.0	27.1

预处理宏，代替了变量（我们称这些宏为类对象宏）和函数（我们称之为类函数的宏）。在现代C语言中，类对象宏常常可以被枚举替代，类函数宏被内联函数代替。两种替代品都只作用于C的块（block），因此，比宏安全，因为宏作用于整个编译单元。表15-5最后三个指标衡量了类功能和类对象宏的数量。鉴于可替代的选择和与宏有关的危险，理想值越低越好。

15.3.5 数据组织

最后一组测量数据关注每个内核的（内存）数据组织。C代码数据组织质量的衡量标准，取决于标识符的作用域和数据结构的使用。

与许多现代语言相反，C几乎没有提供机制来控制命名空间污染。函数只能有两个作用域（文件和全局），宏在被定义的编译单元内有效，聚合标记往往全局有效。因为可维护性的缘故，大规模系统（如这四个操作系统）需要谨慎地使用现有机制，以控制大量的标识符冲突。

图15-15显示了每个函数在开始处可见的标识符和宏的平均数量，来显示C文件的命名空间污染的程度。有大约10 000个标识符在任何系统的任何一点都可见，很明显，命名空间污染是C代码问题。然而，FreeBSD比其他系统情况好点，WRK最糟。

表15-6前三个指标研究每个系统如何处理其最稀缺的命名资源，全局标识符。可以通过尽量减少全局标识符的数量来减少空间污染。此外，尽量减少全局变量作为操作数的百分比可以降低耦合，减轻代码的理解难度（全局标识符可在组成该系统的任何文件中）。最后一个指标统计了可以在静态范围内定义，却被定义为全局的全局标识符，因为它们只有一个文件在访问而已。对

应的SQL查询计算了只在一个文件中存在的全局标识符（链接单元）百分比：

```
select 100.0 * (select count(*) from
  (select  TOKENS.eid from TOKENS
    left join IDS on TOKENS.eid = IDS.eid
    where ordinary and lscope group by eid having min(fid) = max(fid) ) static) /
  (select count(*) from IDS)
```

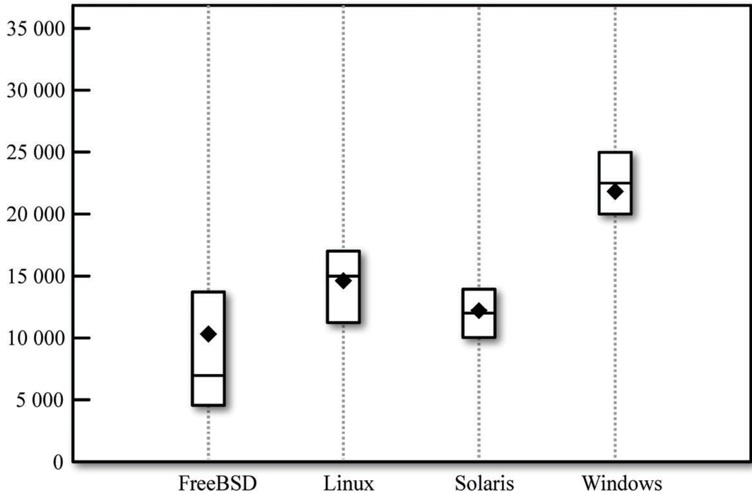


图15-15　C文件中命名空间污染平均水平

接下来的两个指标着眼于变量声明和文件范围内的操作数。它们都比全局变量危害小，但比块作用域中的变量危害大。

表15-6　数据组织指标

指　标	理 想 值	FreeBSD	Linux	Solaris	WRK
全局变量百分比	↓	0.36	0.19	1.02	1.86
全局操作数百分比	↓	3.3	0.5	1.3	2.3
错误的全局标识符百分比	↓	0.28	0.17	1.51	3.53
文件内变量百分比	↓	2.4	4.0	4.5	6.4
文件内操作数百分比	↓	10.0	6.1	12.7	16.7
每个typedef或是聚合中的变量	↓	15.13	25.90	15.49	7.70
每个聚合或是枚举的数据元素	↓	8.5	10.0	8.6	7.3

最后两个有关数据组织的指标提供了一份代码中抽象机制使用的衡量标准。类型定义和聚合是C语言中两个主要的数据抽象机制。因此，统计类型定义与聚合函数的数量揭示了每个系统对于抽象化机制的使用程度。

```
select ((select count(*) from IDS where ordinary and not fun) /
  (select count(*) from IDS where suetag or typedef))
select ((select count(*) from IDS where sumember or enum)  /
```

```
(select count(*) from IDS where suetag))
```

最后这些语句统计了每个聚合或是枚举中数据元素的数量及其与整个数据元素的关系。这像是Chidamber和Kemerer的面向对象的每类中加权方法（WMC）度量[4]之于代码一样。较高的值可能表明一个结构试图存储太多不同的元素。

15.4　成果和结论

有两种统计数据：一种是你查阅的；一种是你创造的。

——Archie Goodwin①

表15-7总结了我的实验结果。单元内的“+”表示优秀，“–”表示落后。表里面的数据有局限性。首先，指标的权重还是根据它们的重要性校准的。此外，指标是不是和功能无关，这一点也不清楚，或者说他们是否完整地或是很有代表性地体现了C代码质量情况，也不是很清楚。最后，我主观地加入了+ / -，希望能清楚地指出几个指标里不同系统的结果。

表15-7　结果总汇

指　　标	FreeBSD	Linux	Solaris	WRK
文件组织				
C文件长度			–	–
头文件长度		+		–
C文件中的全局函数			–	–
头文件中的定义结构				–
目录结构		+		
每个目录中文件的个数		–		
每个C源文件的头文件数量				
平均文件中的结构复杂度	–		+	
代码结构				
扩展圈复杂度		+		–
每个函数的语句数		+		
Halstead复杂度		+		–
文件级耦合		–		
全局耦合		+		
全局函数百分比		+		–
严格结构的函数百分比	–			+
带有标签的语句百分比		–		+
函数参数平均数量				
平均最大层级嵌套深度			–	–
每条语句中的单词数				

① Archie Goodwin，虚构小说中人物。此句话出自侦探小说家Rex Stout的小说*Death of a Doxy*（1996）。——译者注

（续）

指　　标	FreeBSD	Linux	Solaris	WRK
重复代码中的标记数目	−	−	+	
函数中的评价结构复杂度	+	−		
代码风格				
全局标识符长度				+
聚合标识符长度				+
对齐行数百分比			+	−
typedef对齐百分比	−	−		+
聚合对齐百分比	−	−	−	+
每行的字符数				
操作数中的数值常量百分比		−	+	+
不安全类函数宏百分比			−	
C文件中的注释密度		−		+
头文件中的注释密度		−		+
注释中的拼写错误百分比				+
注释中独特的拼写错误百分比				+
预处理				
函数中预处理扩张	−		+	
文件级预处理扩张		−		−
头文件中的预处理指令百分比		−	−	+
头文件中非_#include的指令百分比	−		+	
函数中的预处理器命令百分比	−		+	
函数中的预处理条件百分比	−	+	+	
定义函数内类函数的宏百分比		+		−
单独标识符中的宏百分比	−		+	+
标识符中的宏百分比	−		+	
数据组织				
文件中命名污染平均水平	+			−
全局变量百分比		+		−
全局操作数百分比	−	+		
错误的全局标识符百分比		+		−
文件内变量百分比	+			−
文件内操作数百分比		+		−
每个typedef或是聚合中的变量		−		+
每个聚合或是枚举的数据元素		−		+

然而，通过观察标记的分布和集中情况，我们可以得出一些重要而可信的结论。从前面章节的详细列表和表15-7中，我发现了一个有趣的现象，就是系统之间不同值的相似性。在各个领域和许多不同的指标中，四个有很大不同开发流程的系统的得分是相似的。最起码，这个结果可以

说明一个庞大而复杂软件构件的结构和内部质量，首先代表的是艰巨的工程建设需求，而流程的影响最多也只是边缘化的。如果你打算建立一个真实世界的操作系统，汽车的电子控制单元，空中交通管制系统，或火星着陆探测器上的软件，不管你是管理一个专有软件开发团队或运行一个开源项目：你不能忽视质量。这并不意味着流程是无关紧要的，但它确实意味着与需求兼容的流程会产生大致相似的结果。在建筑领域的这一现象被称做“形式追随功能”[32]。

你也可以从某些部分的集中情况得出有趣的结论。Linux代码结构的各种指标都不错，但在代码风格上落后。这可能是由于聪明而又积极地程序员谁都没有注重代码风格的细节。与此相反，WRK在代码风格中的高得分，代码结构得分低，却可能是因为太过注重细节，却没有充分自由地发挥创造力去发展技术、设计模式和工具来克服大规模软件的复杂度。

OpenSolaris在预处理方面的得分领先可能是因为它的编程规范。使用预处理器的问题很明显，但是诱惑也很大。比如说使用预处理器来构造域相关的编程构架是很方便的。使用预处理器也很容易为不同的平台设定不同的编译命令，解决了可移植性问题。但长期来看，这些都有问题。我们可以假设像Sun这样的公司是不鼓励程序员用预处理器的。

还有一个有趣的现象表现在FreeBSD在预处理方面的不良表现。这可能是因为狂热的FreeBSD开发者认为读代码的人都是开发人员或者至少跟写代码的人一样聪明。不过，FreeBSD命名空间污染较少，也许也是因为使用预处理器建立了保守的数据结构访问权限。

尽管人们对于开放式或封闭源代码的开发效率意见并不统一，但从我们的实验结果来开，并没有明显的获胜者或是失败者。具有商用血统的OpenSolaris各方面的得分比较均衡。WRK负面的得分最高，OpenSolaris正面得分位于倒数第二位。开源的操作系统中，虽然FreeBSD负面得分最高，正面得分最低，但是Linux在正面得分中处于第二位。我们从中最能得出的结论就是：开源的开发方式得到的代码并不比专有软件的开发方式得到的代码质量高。

15.5 致谢

我要感谢微软、Sun、FreeBSD和Linux社区的成员，他们提供各自的源代码给我做分析和实验。我也要感谢Fotis Draganidis、Robert L. Glass、Markos Gogoulos、Georgios Gousios、Panos Louridas和Konstantinos Stroggylos的帮助、评价和对早期草案的建议。这项工作的部分资金由欧盟第六期科研架构计划资助，合同号IST- 2005 -033331“开源软件质量监测（SQO- OSS)”。

自2003年以来，我一直是FreeBSD项目的源代码提交者。我被邀请参加过三次微软赞助的学术活动，这四个系统也用了十多年的时间。

15.6 参考文献

[1] [Cannon et al.] Cannon, L.W., and others. Recommended C style and coding standards. Archived by WebCite® at *http://www.webcitation.org/5svpriKD3*.

[2] [Cant et al. 1995] Cant, S.N., D.R. Jeffery, and B.L. Henderson-Sellers. 1995. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology* 37(7): 351-362.

[3] [Capiluppi and Robles 2007] Capiluppi, A., and G. Robles, ed. 2007. *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*. IEEE Computer Society.

[4] [Chidamber and Kemerer 1994] Chidamber, S.R., and C.F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20(6): 476-493.

[5] [Coleman et al. 1994] Coleman, D., D. Ash, B. Lowther, and P.W. Oman. 1994. Using metrics to evaluate software system maintainability. *Computer* 27(8): 44-49.

[6] [Cusumano and Selby 1995] Cusumano, M.A., and R.W. Selby. 1995. *Microsoft Secrets*. New York: The Free Press.

[7] [Dickinson 1996] K. Dickinson. 1996. Software process framework at Sun. *StandardView* 4(3): 161-165.

[8] [Feller 2005] FellerJ., ed. 2005. *5-WOSSE: Proceedings of the Fifth Workshop on Open Source Software Engineering*. ACM Press.

[9] [Feller and Fitzgerald 2001] Feller, J., and B. Fitzgerald. 2001. *Understanding Open Source Software Development*. Reading, MA: Addison-Wesley.

[10] [Feller et al. 2005] Feller, J., B. Fitzgerald, S. Hissam, and K. Lakhani, ed. 2005. *Perspectives on Free and Open Source Software*. Cambridge, MA: MIT Press.

[11] [Fitzgerald and Feller 2002] Fitzgerald, B., and J. Feller. 2002. A further investigation of open source software: Community, co-ordination, code quality and security issues. *Information Systems Journal* 12(1): 3-5.

[12] [The FreeBSD Project 1995] The FreeBSD Project. 1995. Style—Kernel Source File Style Guide. The reeBSD Project. FreeBSD Kernel Developer's Manual: style(9). Archived by WebCite® at *http://www.webcitation.org/5svq73uER*.

[13] [Gill and Kemerer 1991] Gill, G.K., and C.F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering* 17(12): 1284-1288.

[14] Glass 1999] R.L. Glass. 1999. Of open source, Linux...and hype. *IEEE Software* 16(1): 126-128.

[15] [Halstead 1977] M.H. Halstead. 1977. *Elements of Software Science*. New York: Elsevier New Holland.

[16] [Harbison and Steele Jr. 1991] Harbison, S.P., and G.L. Steele Jr. 1991. *C: A Reference Manual*, Third Edition. Englewood Cliffs, NJ: Prentice Hall.

[17] [Henry and Kafura 1981] Henry, S.M., and D. Kafura. 1981. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* SE-7(5), 510-518.

[18] [Hoepman and Jacobs 2007] Hoepman, J.H., and B. Jacobs. 2007. Increased security through open source. *Communications of the ACM* 50(1), 79-83.

[19] [Izurieta and Bieman 2006] Izurieta, C., and J. Bieman. 2006. The evolution of FreeBSD and Linux. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, 204-211.

[20] [Jørgensen 2001] N. Jørgensen. 2001. Putting it all in the trunk: Incremental software development in the FreeBSD open source project. *Information Systems Journal* 11(4): 321-336.

[21] [Kernighan and Plauger 1978] Kernighan, B.W., and P.J. Plauger. 1978. *The Elements of Programming Style*, Second Edition. New York: McGraw-Hill.

[22] [Kuan 2003] J. Kuan. 2003. Open source software as lead user's make or buy decision: A study of open and closed source quality. Paper presented at the second conference on the Economics of the Software and Internet Industries, January 17 – 18, in Toulouse, France. Archived by WebCite® at *http://www.webcitation.org/5svpriKD3.*

[23] [Li et al. 2006] Li, Z., S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-miner: Finding copy-paste and related bugs in

large-scale software code. *IEEE Transactions on Software Engineering* 32(3): 176-192.

[24] [McCabe 1976] T.J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* 2(4): 308-320.

[25] [McConnell 2004] , S.C. McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction*, Second Edition. Redmond, WA: Microsoft Press.

[26] [Parnas 1972] D.L. Parnas. 1972. On the criteria to be used for decomposing systems into modules. *Communications of the ACM* 15(12): 1053-1058.

[27] [Paulson et al. 2004] Paulson, J.W., G. Succi, and A. Eberlein. 2004. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering* 30(4): 246-256.

[28] [Polze and Probert 2006] Polze, A., and D. Probert. 2006. Teaching operating systems: The Windows case. *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*: 298-302.

[29] [Rigby and German 2006] Rigby, P.C., and D.M. German. 2006. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria. Archived by WebCite® at *http://www.webcitation.org/5svoPB5t5*.

[30] [Salus 1994] P.H. Salus. 1994. *A Quarter Century of UNIX*. Boston, MA: Addison-Wesley.

[31] [Samoladas et al. 2004] Samoladas, I., I. Stamelos, L. Angelis, and A. Oikonomou. 2004. Open source software development should strive for even greater code maintainability. *Communications of the ACM* 47(10): 83-87.

[32] [Small 1947] Small,H.A., ed. 1947. *Form and Function: Remarks on Art by Horatio Greenough*. Berkeley and Los Angeles: University of California Press.

[33] [Sowe et al. 2007] Sowe, S.K., I.G. Stamelos, and I. Samoladas, ed. 2007. *Emerging Free and Open Source Software Practices*. Hershey, PA: IGI Publishing.

[34] [Spencer and Collyer 1992] Spencer, H., and G. Collyer. 1992. #ifdef considered harmful or portability experience with C news. *Proceedings of the Summer 1992 USENIX Conference*: 185-198.

[35] [Spinellis 2003] D. Spinellis. 2003. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering* 29(11): 1019-1030.

[36] [Spinellis 2006] D. Spinellis. 2006. *Code Quality: The Open Source Perspective*. Boston, MA: Addison-Wesley.

[37] [Spinellis 2008] D. Spinellis. 2008. A tale of four kernels. *ICSE '08: Proceedings of the* 30th *International Conference on Software Engineering*: 381-390.

[38] [Spinellis 2010] D. Spinellis. 2010. CScout: A refactoring browser for C. *Science of Computer Programming* 75(4): 216-231.

[39] [Spinellis and Szyperski 2004] Spinellis, D., and C. Szyperski. 2004. How is open source affecting software development? *IEEE Software* 21(1): 28-33.

[40] [Stallman et al. 2005] Stallman, R., and others. 2005. GNU coding standards. Archived by WebCite® at *http://www.webcitation.org/5svos1oZq*.

[41] [Stamelos et al. 2002] Stamelos, I., L. Angelis, A. Oikonomou, and G.L. Bleris. 2002. Code quality analysis in open source software development. *Information Systems Journal* 12(1): 43-60.

[42] [Stol et al. 2009] Stol, K.J., M.A. Babar, B. Russo, and B. Fitzgerald. 2009. The use of empirical methods in open source software research: Facts, trends and future directions. *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*: 19-24.

[43] [Tanenbaum 1987] A.S. Tanenbaum. 1987. *Operating Systems: Design and Implementation*. Englewood Cliffs,

NJ: Prentice Hall.

[44] [Torvalds and Diamond 2001] Torvalds, L., and D. Diamond. 2001. *Just for Fun: The Story of an Accidental Revolutionary*. New York: HarperInformation.

[45] [von Krogh and von Hippel 2006] von Krogh, G., and E. von Hippel. 2006. The promise of research on open source software. *Management Science* 52(7): 975-983.

[46] [Yu et al. 2004] Yu, L., S.R. Schach, K. Chen, and J. Offutt. 2004. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Transactions on Software Engineering* 30(10): 694-706.

[47] [Yu et al. 2006] Yu, L., S.R. Schach, K. Chen, G.Z. Heller, and J. Offutt. 2006. Maintainability of the kernels of open source operating systems: A comparison of Linux with FreeBSD, NetBSD and OpenBSD. *Journal of Systems and Software* 79(6): 807-815.

第16章

码　语　者

Robert DeLine

由于大众文化的渲染，人们总以为程序员都是窝在小隔间办公室中不懂怎么和别人交往的书呆子，就像漫画《呆伯特》的主角Dilbert和Wally，或者英剧《IT狂人》（*The IT Crowd*）中的Roy和Moss那样。有了这种印象之后，人们就会觉得这些极客们只有讨论柯克和皮卡尔[①]到底谁更优秀的时候才会开口。

可真实的情况并非如此，很多程序员都会花上很大一部分的时间离开办公桌相互聊天交流。在这一章中，我们将介绍数个关于程序员如何安排工作时间的研究，并将看到，交流是程序员工作的主要组成部分。这里的交流并不是在饮水机旁边站着闲聊，而是帮助程序员们相互交谈以了解项目当前的状况、进行协调活动以及相互分享知识的必要手段。我们不应该把这类交流看做妨碍团队运作的绊脚石，反倒是应该将其看成是保证团队平稳工作的润滑剂。

16.1　程序员的一天

那么，普通程序员们每天是如何安排自己的工作时间的呢，我们又如何知道呢？

从很早以前开始，学者们就一直在研究工人们如何分配时间并加以改善，这已经成为了一种传统。20世纪初的时候，像Frank Gilbreth和Lillian Gilbreth这样的前沿研究者就已经由于写了《儿女一箩筐》而声名大振，他们介绍了一种“时间与动作研究”的方法，用于降低工厂工人以及办公室人员在执行重复任务时所需要花的时间和精力。使用这种方法的研究人员将配备秒表、写字板和摄像头，并对工人的正常工作活动进行观察。研究人员会记录下每个身体动作的时间，并看能不能找出一些更快、更准确和更轻松的新的动作来代替。例如Gilbreths发现外科医生在手术的时候常常会去摸索寻找手术工具，然后发明了我们现在熟知的规矩，即外科医生需要工具的时候就让助理护士来传递。这样的创新不但缩短了手术的时间，也避免了很多错误。

当然我们必须看到，软件开发和工厂以及手术是不同的。时间与动作研究主要关注的是形体动作，而软件开发主要靠的是大脑。不过，这个研究方法对软件开发却同样适用。

在20世纪90年代初期，Perry、Staudenmayer和Votta进行了一项研究，使用的方法正是时间与

① 柯克和皮卡尔均为《星际旅行》中的星舰舰长。——译者注

动作研究，对象是一组实时交换系统的程序员[5]。由于当时还没有一套成文的方法来研究程序员的工作，所以他们尝试了两种不同的方法：日记研究和观察研究。

16.1.1　日记研究

首先，研究人员们要求参与者们在一年内每天填写一份工作日记，记录他们每日的工作活动。工作时间中的每个小时都有对应的格子，在这些格子中总结他这个小时做了些什么。参与者将在每个格子中写下他所分配到的任务类型（如规划、需求、设计、编码和测试等）。如果参与者并没有进行任务，他将写下原因（比如说，有其他更高优先级的任务、必须的资源不可用所以等待或者个人选择做其他事情等）。出乎意料的是，4个小组中有13个程序员都愿意参与这个实验。

基于这个研究的数据，他们得出的最重要结论是参与者只花了约40%的时间来进行分配到的任务。在剩下的时间中他们要么就在等待，要么就在做其他事情。研究人员们注意到，由于常常会被迫等待，程序员们一般都会保证手上同时有两件工作。这样的话，如果进行一个任务受阻等待的时候，他们就可以切换到另外一项任务上去。

当然，由于参与者是自己记录自己的时间使用情况，所以这样的日记可能并不会特别准确。虽然我们可以假设每个参与者都敬业地在每个小时的整点时刻填写日记，但是用小时作为单位也是非常粗略的——一个小时之内可能发生的事情太多了！所以，为了核查日记的准确性并挖掘更详细的数据，研究人员又进行了第二次研究，在这次研究中，他们将跟踪观察程序员们并记录下他们的时间使用情况。

16.1.2　观察研究

在第二次研究中，三位研究员直接观察了5个参与者的日常工作。在工作时间中，他们一直跟随着这些程序员，一般每天8到10个小时，每人分别观察了5天，共获得了多达300小时的数据。在受观察期间，研究人员要求被观察者把自己当做一个实习的学生。在研究人员看不明白程序员正在做的事情的时候，他们就会主动问问题，比如“你现在在做什么？”[6]。这样的观察数据比起日记来说要仔细得多，时间的单位也精细到了3分钟。

从观察的数据来看，程序员每天大概会花75分钟的时间来进行非正式的沟通，沟通方式则是email、电话、语音邮件或者面对面的谈话。这75分钟*并不包含*在定期的设计评审、代码评审、内部通气会等会议中，很明显这些会议也将包括很多的沟通。除了邮件列表中的广播信息之外，最常见的沟通方式是面对面地谈话，这种方式的频率是其他方式的两到三倍。程序员们每天平均会跟7个不同的人说话。在研究中发现谈话对象数量最多的是17人。这些交流一般比较短，其中68%都少于5分钟，但是超过半小时的电话或者长达一小时的面对面交谈也不罕见。很明显，说了不少话啊！

16.1.3　程序员们是不是在挣表现

如果我们天真地以为程序员们只有在敲键盘的时候才能出成绩，那么这些结果可能看起来很

危险。毕竟，研究表明程序员们大部分的时间都不是花在任务上，而其中很大一部分是用来和其他程序员们交谈。我们可能会问这些研究是不是哪里出了错。

有种说法是日记研究法太依赖自我汇报的数据了。为了解决这个问题，研究人员们使用了日记和观察两种方法。所以说，有很多天的数据既包括了自我汇报的日记又有研究人员的观察数据。也许你已经想到了，程序员们对任务时间的估算有差别，有时候会多汇报有时候会少汇报。但是，差距一般不会很大：平均只有2.8%的时间是多汇报的。在很多情况下，少回报的原因都是因为干扰和等待而忽略了时间。

当然，我们还可以怀疑观察到的数据。由于研究人员会一直注视着程序员们，我们可能会感觉程序员们是在观察期间“做样子”，甚至“挣表现”。实际上，这种现象在实验心理学中非常常见，称之为：霍桑效应。

在20世纪20年代后期，研究人员在位于芝加哥的霍桑电子工厂做了一些实验，想要看看改善车间照明能不能提高工人的效率。在几个星期的时间里，他们尝试了多种不同的照明水平。有趣的是，无论他们对照明的调整再小，工人的效率都会提高！但是研究一结束，效率就立刻恢复到原来的水平了。这些实验有一个共同点，那就是工人们都知道自己是实验对象。今天，我们用“霍桑效应”来指当被观察者知道自己是研究对象的时候，会改善自己的行为的现象。

那么，这个研究中的参与者们是否收到了霍桑效应的影响呢？由于这个影响是不可避免的，所以答案是肯定的，不同的只是影响的程度问题。所以，我们更应该问：霍桑效应在多大程度上影响了观测数据的准确性？由于研究人员们知道这个效应，所以他们在设计研究的时候就考虑了如何来减轻它的影响。首先，在日记研究中的很多参与者并没有参与观察研究，这样他们就可以知道被观察的程序员和没被观察的程序员有什么不同。其次，观察日期并不是提前安排好的。由于研究人员每天都可能出现，程序员们就无法故意把自己的工作安排得“很有趣”以应对观察。在这次研究中，随机选择和事先安排的差距并不大。最后，在两次研究中，数据都是完全匿名的，这样就降低了程序员们“挣表现”的可能性，因为他们的同事、经理和其他人无法得知他们的结果。

16.2　说这么多有什么意义

由于程序员们很大一部分的工作时间都花在了谈话上，你可能会在想他们到底在谈些什么。我和我的同事们最近在微软公司进行的两次研究对这个问题进行了一些探索。通过问卷调查、访谈和直接观察的方式，我们了解到，大部分的谈话都是在寻找信息——即为编程任务相关的问题寻求答案。这些研究使我们了解了很多程序员偏好面对面交谈的原因，还让我们知道了最容易出现的问题究竟是什么。

16.2.1　问问题

在2006年开始的第一项研究中，我们随机抽取了157名程序员进行问卷调查，并随后对他们中的11人进行了访谈。我们询问了他们的日常工作活动，尤其是他们认为最困难的工作[4]。他们

关于时间分配的反馈和Perry等人的研究一致。花在沟通上的时间的平均比率要比其他任何活动都要高。问卷调查还要求受访者估计他们使用各种沟通方式的时间比率，并按照他们的想法对这些沟通方式的效率进行评价。结果如图16-1所示。这些结果仍然和Perry等人的研究一致，受访者大部分的时间仍花在即兴的面对面谈话上。他们也将面对面交流评为最有效的沟通方式。

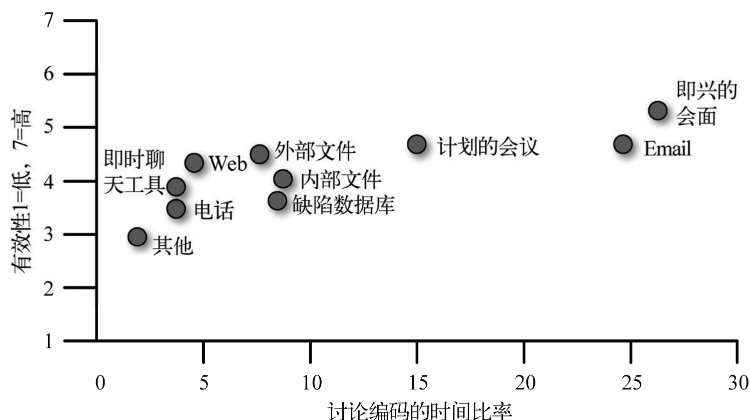


图16-1　不同沟通方式的使用和有效性评估

在访谈中，我们询问了他们在日常工作中会碰到哪些难题。他们都有一个共同的难题，那就是常常会因为缺少一些必需的信息而不得不卡在哪里。他们通常认为穿过走廊来和同事交谈不但效率高，而且有时要想获得信息就只能靠这个办法。

下面这个故事取自我们的访谈之一（参见参考文献[4]），它说明了在调试环节中，沟通也是编程活动的重要组成部分。

> 这个开发人员通过bug跟踪工具中得到了一个bug，他重现了这个错误，打开了某个网址，并确保这个网址如bug报告所称会报“500 - Internal Error”错误。然后，他将Visual Studio的调试器连接到Web服务器，并设置为出现异常即暂停的状态，并在此重现错误，然后Visual Studio告诉他问题是空指针异常。在查看了调用栈窗口之后，他认为某个函数可能在生成错误的值。他用Emacs打开了这个函数，并使用了ctags.exe浏览这个函数的调用者。然后，他转回了Visual Studio的调试器，并改变了运行时的值，看看有什么效果。他做了修改，并重新编译，然后发现错误还在。最后，他进一步浏览了调用栈，发现错误的值都是由一个对象生成的，然后他又发现了另一个对象，最后发现了第三个具备互斥保护的对象。
>
> 而这个时候，他不得不去查看并不“属于”他或者由他来维护的代码，但是看不明白。另一个开发人员在开发某个功能的时候修改过代码，而他知道这个开发人员应该能看懂这些代码。然后他到了这个开发人员的办公室，打断那个开发人员，两人开始讨论这段代码的设计思路。
>
> 随后他返回了自己的办公室，并根据得到的信息作出了修改，却发现修改无效，这

下老问题没解决，他又在这段不属于他的代码中发现了新问题。他又到了那个开发人员的办公室，这个开发人员告诉他造成问题的功能实际上是由第三个开发人员来负责的。他们一起到了第三个开发人员的办公室，但是他去吃午饭了。第一个开发人员由于无法继续这个任务，所以开始进行另一个任务。午饭之后，两个开发人员又重新到了第三个开发人员的办公室，然后讨论了功能的正确行为，并最终把第一个开发人员的问题交给了第三个开发人员进行修复。

这个故事充分地展现了我们在访谈中多次听到的问题。当程序员被困在某个他不明白的问题上时，他就会先做一些“份内的事”，再问同事帮忙。也就是说，他会先花时间了解代码，以免问出太不靠谱的问题。在面谈之后，我们又组织了一次有187个程序员参加的后续问卷调查，以便能了解更多关于这个“份内事”的信息[4]。在这次调查中，我们发现在做“份内事”的过程中，受访者平均会花42%的时间来分析源代码，20%的时间来使用编译器，16%的时间来分析提交注释和版本比较，9%的时候来分析结果，8%的时间来使用调试和跟踪语句以及5%的时间使用其他方法。

16.2.2 探寻设计理念

一般来说，仅仅分析代码及其行为并不能透彻理解它的意义。在前一节提到的故事中，受访的程序员在同事那里学习了代码背后的设计理念——这样的信息通常是没有记录的，只存在于团队的集体记忆之中。正如某位受访程序员所说的那样：“很多设计信息，都是在人们的大脑中保存着的。”实际上，在我们第一次调查中，“了解代码的设计理念”是排名最高的问题，有66%的人都认为他们遇到了这个困难。

那么到底“设计理念”是什么意思？在后续问卷调查中，我们询问受访者要了解哪方面的设计理念最为困难。在187个受访者中，有82%认为最难理解的是“为什么代码是按这种方式来实现的”，有73%的人认为最难的是“是否这段代码只是暂时的解决方案”，有69%认为是“代码是如何工作的”，有62%认为是“这段代码想要完成什么功能”。也就是说，设计理念就是代码“背后的故事”，有时候是关于为什么这样写，有时候是关于为什么做出这样的取舍。

16.2.3 工作的中断和多任务

再回到我们的访谈故事。当这个程序员遇到设计理念问题的时候，他穿过走廊找同事面谈，先找了一个，后来又找了另一个。这样的选择和我们在图16-1中所示的沟通偏好以及Perry等人的观察一致。问题是，每次即兴的面谈都至少会让一个人中断他的工作。工作的中断又会造成任务的切换。比如说，被中断的人可能会停止写代码，并开始为提问的人寻找答案。在我们的问卷调查中，排名第二的问题是“由于队友或者管理者的请求而不得不切换任务”，共有62%的受访者认为这是个难题。

在故事中我们还看到这个程序员在第一次尝试时曾受阻，因为他的同事不在。结果就是，这个程序员也切换了任务，以免没事做。随后，当那个同事回来的时候，他们一起继续进行了原来

的任务，即确定了程序的正常行为以及谁负责实现。Perry等人也观察到了同样的任务切换行为。对于被中断的人来说，这样的面谈就意味着效率的流失，但是对于问问题的人来说，如果想要保持高效，这样的面谈却是必不可少的。

总的来说，当程序员们执行任务的时候，常常会出现各种问题。当他们“恪尽职守”地翻阅代码却仍无法找出答案的时候，他们常常会去问同事。这些问题中最难解决的是设计理念的问题。这种问题的答案常常只存在于“人的大脑中”。但是设计理念是不是唯一的问题呢？在程序员工作的时候还会遇到什么其他的问题呢？

16.2.4　程序员都在问什么问题

为了了解程序员们在日常工作中会问的问题，我们又在微软公司做了第二次研究[3]，使用了一套和Perry等人使用的观察研究法非常相似的方法。我们观察了17个程序员的日常工作，每人90分钟。就像Perry等人那样，我们也要求程序员把我们当作实习的学生，可以在不明白他们在做什么的时候问问题。

我们还使用了心理学家和社会学家们所谓的“想出声来”（think-aloud）的方法。也就是说，要求程序员不断地跟我们说话，说出他们的每个想法，并描述在进行的每个动作。下面是一个虚构的例子。

> 好吧，我正在找打开数据库的代码，所以我按了Ctrl-F来打开搜索框。我正输入“open”。该死，啥也没有。我再试试“database”。好了，我找到代码了。它被称之为“access database”，不是“open”，所以我才没找到。

虽然这样的碎碎念看起来好像有点令人不舒服，有时甚至让人讨厌，但是参与者们很快就适应了。如果没有“想出声来”的办法，我们就不可能理解程序员的精神状态，更无从了解他们常常在疑惑的问题。

参与者就这样不断地说着，而我们则会用一个小的数字显示式时钟和一个笔记本记下他们在每分钟里说过的所有话和做过的所有事情。当他们离开座位去问同事问题的时候，我们也跟着并记录下整个对话。记下了这些数据之后，我们又对它们进行了分析和分类，如表16-1所示。

表16-1　实验观察到的程序员需求，按频率排列

信息类型	持续时间		频率及寻找的结果
	平均	最多	找到■ 推迟□ 放弃⊠ 未观察到 -
我的同事们都做了什么	1	11	■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■□□□□□□
是什么代码导致了这个程序状态	2	21	■■■■■■■■■■■□□□□□□□□□□□□□□□□□□⊠ -
在什么情况下会出现这个错误	2	49	■■■■■■■■■■■■■■■□□□□□□□□□□□⊠ -
这个程序本来是做什么的	1	21	■■■■■■■■■■■■■■■■■■■■■■■□□□□
别人对我所依赖的资源进行了哪些变动	1	9	■■■■■■■■■■■■■■■■■■■□□□□□⊠
什么代码可能造成了这个行为	2	17	■■■■■■■■■■■■■□□□□□□⊠⊠ -
我如何来使用这个数据结构（或者函数）	1	14	■■■■■■■■■■■■■■■■■■■□⊠

（续）

信息类型	持续时间		频率及寻找的结果
	平均	最多	找到■ 推迟□ 放弃⊠ 未观察到 -
这个代码为什么是这样实现的	2	21	■■■■■□□□□□□□⊠ - - - - -
这个问题是否值得修复	2	6	■■■■■■■■■■■■■■□
这样修改会造成什么影响	2	9	■■■■■■■■■■■■■■
这段代码的目的是什么	1	5	■■■■■■■■■□⊠⊠
这个代码和哪些代码有静态关联	1	7	■■■■■■■■■□
这算不算一个问题	1	2	■■■■■■■
我有没有遵守团队的惯例	7	25	■■□□□□
这个故障应该是什么样子	0	2	■■■■■
这次提交包含哪些修改	2	3	■■■□
我如何协调这段代码和其他代码	1	4	■■⊠ -
这个问题有多难解决	2	4	■■■
我们有哪些选择来实现这个程序行为	2	2	■■
哪些信息与我的任务相关	1	1	■■

表中的信息是我们从听到的具体问题中概括出来的。对于每种信息需求，我们还将展示两类信息。首先，“持续时间”栏展示了我们观察到的程序员寻求这个信息需求答案的平均和最多分钟数。其次，“频率和结果栏”中的每个符号则代表我们观察到的一次信息需求：黑方框意味着成功找到答案，白方框意味着推后了一段时间，带X的盒子意味着放弃寻找，而短横线则代表结果未知（因为在问题解决之前观察的时间就到了）。

程序员在哪里找答案？表16-2展示了我们观察到的程序员的信息来源。和前面的研究结果一致，程序员们最常询问同事（coworker）以获得信息。而其他的信息来源，按照最常用到最少用排列分别是：

(1) 各种团队或者公司专用的工具（tool）；

(2) 程序员自己的直觉、逻辑推论或者记忆（brain）；

(3) Bug数据库（bug）；

(4) 调试器（dbug）；

(5) 源代码、其注释或者历史（code）；

(6) 除了规格文档之外的文档（docs）；

(7) Email；

(8) 规格文档（specs）；

(9) 程序执行的日志文件（log）；

(10) 即时通信工具（im）。

除了调查最常见的问题之外，我们还调查了最让人痛苦的问题，也就是那些搜寻答案的时间最长或者没能解决（推迟或者放弃了）的问题。根据这个标准，最让人痛苦的7大信息需求分别是下列问题。

(1) 是什么代码导致了这个程序状态？（61%未解决，最多21分钟。）

(2) 这个代码为什么是这样实现的？（44%未解决，最多21分钟。）

(3) 在什么情况下会出现这个错误？（41%未解决，最多49分钟。）

(4) 什么代码可能造成了这个行为？（36%未解决，最多17分钟。）

(5) 别人对我所依赖的资源进行了哪些变动？（24%未解决，最多9分钟。）

(6) 这个程序本来是做什么的？（15%未解决，最多21分钟。）

(7) 我的同事们都做了什么？（14%未解决，最多11分钟。）

我们观察到，当程序员遇到上面这些问题的时候，都去问了同事，但是也利用了其他的信息源来寻找答案（这些通常是他们“恪尽职守”的一部分）。

所以，除了知道开发人员常常相互问问题之外，我们还知道了哪些问题问得最多，哪些问题最难解决。

表16-2　程序员寻求答案的信息来源的使用频率

	coworker	tool	brain	bug	dbug	code	docs	email	specs	log	im	总计
我的同事们都做了什么	20	8						13			2	43
是什么代码导致了这个程序状态	1	3	3	3	16	2				3		31
这个程序本来是做什么的	9						5	1	13			28
在什么情况下会出现这个错误	8	3	5	8	2	1						27
别人对我所依赖的资源进行了哪些变动	6	12		2		1		4				25
什么代码可能造成了这个行为	5		4	4	2	1			1	4	1	22
我如何来使用这个数据结构（或者函数）	4					5	11		1			21
这个代码为什么是这样实现的	2	2	4	1	2	8						19
这个问题是否值得修复	12		1	1				2				16
这样修改会造成什么影响	13									1		14
这段代码的目的是什么		2	5		2	2	1		1			13
这个代码和哪些代码有静态关联		8	2					1				11
这算不算一个问题	1			5						1		7
我有没有遵守团队的惯例		2	1				2					5
这个故障应该是什么样子				5								5
这次提交包含哪些修改		2	2									4
我如何协调这段代码和其他代码	1					1	2					4
这个问题有多难解决	1			1		1						3
我们有哪些选择来实现这个程序行为			1				1					2
哪些信息与我的任务相关			2									2
总数	83	42	30	30	24	22	22	21	16	9	3	

16.2.5 使用敏捷方法是不是更利于沟通

我们前面描述的行为来自两份研究结果，一份是在20世纪90年代中期在电话公司做的，另一份是在2000年年中在微软做的。这两个公司的产品不同，开发工具不同，而且由于中间隔了十年，他们使用的沟通工具也不同（比如，较早那份研究没有提到手机和即时通信）。但是，两个公司都是使用同一种修改过的传统的瀑布开发流程，即较长的开发周期，以及各个阶段划分清晰，比如规划、写规格文档、编写代码、测试、发布等。如果团队采用了敏捷方法，沟通的行为是否不同？

2006年，Chong和Siino研究了两家位于旧金山湾区的创业公司中敏捷和非敏捷团队之间的区别[1]。其中一个团队（他们称之为“单干团队”）使用传统的开发方法，所有程序员都在各自的隔间里。另一个团队（他们称之为“结对团队”），使用了结对编程的方法，办公室也就是开放式的。总的来说，他们观察到的沟通行为和电话公司以及微软非常相似。两个团队中的程序员都会经常靠沟通来寻求帮助、寻找信息、协调行动或者共享状态。不过，两个团队的沟通性质有所不同，因为开发的方式和工作环境不同。

有的区别是由于办公室环境的物理设置造成的。两个团队都有很大的工作空间，但是结对团队是公共办公室，而单干团队采用的则是小隔间。由于结对团队的环境是开放式的，程序员们可以听到相互之间的交谈，有时候只靠扇扇耳边风就能获得相关的信息，而且还可以随时加入谈话。因为有这个优点，所以结对团队往往会在公开的环境下谈论与工作相关的话题。正好相反的是，单干团队总是试图保持工作场合的安静，因为噪声很容易就会穿透卡座影响到其他人。虽然大家都尽量保持安静，但是单干团队中还是有成员独自在家工作，以避免干扰。（在DeMarco和Lister的名著《人件》中有一个主要的主题就是与此类似的对安静工作环境的需求[2]。）单干团队主要依赖聊天系统进行沟通，这样他们不用动嘴说话就能交换信息，还可以和在家工作的人进行沟通。

Chong和Siino对二者干扰的不同做了深入研究。结对团队的干扰时间总是比单干团队短（结对平均1分55秒，单干2分45秒）。对于结对编程的利用也影响了研究结果。当结对团队遇到问题的时候，通常二人中的一人会去发起讨论并找出答案，而留下另一个人继续工作。此外，在结对团队中，当有人来问问题的时候，其中一个程序员会直接到那个提问的人的办公桌去进行讨论。这意味着回答问题的那个程序员可以自己决定什么时候回答完毕，并回到自己的办公桌。与此相反的是，单干小组和早前介绍的两次研究一样，想问问题的人会走到其他人的办公桌去问问题。这样的话，什么时候回答完毕就由问问题的人来决定了。

总之，这次研究表明在某些层面上，敏捷方法确实可以提供一些沟通上的独特优势。特别需要注意的是对公共办公室的使用，这种办公室的安排方法通常更利于信息和知识的传播，因为大家可以随时扇耳边风，而且这样的安排还会让（至少是公共办公区的）程序员们的谈话更切合工作，并更加简短。结对编程方法的使用使得两人可以分工，一人去回答问题，另一个则继续工作。虽然有这些优点，不过随之而来的还有一些问题。单干团队大量地使用了Email和聊天系统来进行沟通。和谈话的沟通方式不同，这种书面沟通会留下记录，以供以后查阅。大量使用谈话作为沟通方式会不会对敏捷团队的长远发展造成影响还有待研究。

16.3　如何看待沟通

我们可以用一个比喻来帮助你理解本文介绍的各种沟通问题。我们可以把程序员想象成操作系统中的线程调度程序。程序员的工作“算法”如示例16-1所示。

要想弄明白这个比喻，我们首先总览一下操作系统是如何调度线程的。调度器中保存着一个队列的线程，它们都准备好可以运行了。调度器将选择一个线程并运行，直到这个线程被输入所阻塞[①]。被阻塞的线程将被放到等候队列中，直到输入完成。输入是以中断的形式异步抵达的（例如，网络封包抵达系统或者硬盘数据块复制到内存）。当输入抵达时，调度器会找到这个输入所对应的阻塞中的线程，并把这个线程移回活动队列。在处理完中断之后，调度器将继续运行活动队列中的下一个线程。

示例16-1　开发人员的每天就像在做线程调度

```
void beProductiveProgrammer()
{
   while (!quittingTime)
      try {
         var task = readyTasks.pickOne();
         while (!task.isDone && !task.isBlocked)
             task.makeProgress();
         if (task.isBlocked)
             blockedTasks.add(task);
         readyTasks.remove(task);
      }
   catch (Interruption interruption) {
      var info = interruption.informationContent;
      for (var task in blockedTasks)
         if (task.blockedOn(info)) {
             blockedTasks.remove(task);
             readyTasks.add(task);
         }
      }
}
```

这种线程调度的行为和我们在本章介绍的研究中所描述的工作行为惊人地相似。程序员进行编程任务（线程），直到他由于缺乏信息而不能继续进行下去（线程被输入阻塞）。然后她把这个任务放下并开始进行另一个任务，以免没事做。当中断发生的时候（办公室的交谈、打电话、收发电子邮件等），常常也会带来相关的信息（比如之前缺乏的信息）。在程序员了解这个信息之后，就可以恢复此前正等待这个信息的任务。

我们可以天真地认为只有内部的那个循环才是程序员真正的“实效时间”。但是，正如本章介绍的研究表明的那样，这种观点并不符合实际情况。程序员只有在没有问题或者无需交流就能找到答案的时候才会处在内部的那个循环。但是这些研究表明，程序员在日常工作中很多时候都

① 如果线程运行时间太长而又没有受到任何阻塞，调度器就会终止其运行，但是这一点和我们的比喻无关。我们可以用这个来比喻程序员的无聊生活！

会遇到问题，而这些问题都需要沟通才能解决。这些问题的一个主要的原因是对设计理念相关信息的需求，因为这种信息常常只存在于团队的集体记忆中。

我们有没有办法来解决这些问题呢，比如把这些信息更好地记录在文档中？从经济的角度来说，这不是个好主意。开发团队每天会做出上千个决定，这些决定从最小的本地变量的名字，到影响整个产品及团队的需求或者架构。随着项目的发展，有的决定将会不断地被重新提及，而有的决定则是决定了之后就一直保持原状。也就是说对文档进行投资是有相当风险的（或者更直白的说，就是在赌博）。只有当满足下面这些条件的时候，才值得把决定用文档记录下来：

- 有新进的程序员会需要这个信息；
- 这个决定的影响时间将会非常地长，后面进项目的程序员也会受到影响；
- 这样的文档可以回答新进程序员的问题；
- 编写文档的成本低于直接通过沟通回答问题的成本。

这些条件都是猜测性的，我们也并不完全确定。尽管有着这些不确定性，很多团队也还是尽力明智地选择对文档的投入。他们只用文档记录级别最高、受众最广、最为稳定的决策，而剩下的就由沟通来解决了。

16.4 参考文献

[1] [Chong and Siino 2006] Chong, Jan, and Rosanne Siino.2006. Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. *Proceedings of the ACM 2006 Conference on Computer Supported Cooperative Work.*

[2] [DeMarco and Lister 1987] DeMarco, Tom, and Timothy Lister. 1987. *Peopleware: Productive Products and Teams*. New York: Dorset House Publishing Company, Inc.

[3] [Ko et al. 2007] Ko, Andrew J., Robert DeLine, and Gina Venolia.2007. Information Needs in Collocated Software Development Teams.*Proceedings of the International Conference on Software Engineering*.

[4] [LaToza et al. 2006] LaToza, Thomas D., Gina Venolia, and Robert DeLine.2006. Maintaining mental models: a study of developer work habits. *Proceedings of the International Conference on Software Engineering*.

[5] [Perry et al. 1994] Perry, Dewayne, Nancy A. Staudenmayer, and Lawrence G. Votta. 1994.People, organizations, and process improvement. *IEEE Software* 11(4): 36-45.

[6] [Perry et al. 1996] Perry, Dewayne, Nancy A. Staudenmayer, and Lawrence G. Votta. 1996.Understanding and Improving Time Usage in Software Development. In*Process-Centered Environments,* eds. Fuggetta and Wolf.Hoboken, NJ: John Wiley and Sons.

第17章

结 对 编 程

Laurie Williams

结对编程是两个程序员并肩工作在一台电脑前的编程方式。他们在同样的设计、算法、代码或者测试上持续合作。结对编程已经零星地实践了几十年[67]。然而，20世纪末敏捷方法论和极限编程（XP）[4]的出现把结对编程实践推到了众人面前。

在结对编程中，一方叫做“驾驶员”，负责在电脑前打字或者撰写设计。另一方叫做“导航员”，负责许多工作。其中一项是观察驾驶员的工作成果，寻找战术和战略上的缺陷。战术缺陷可能是语法错误、打字错误或调用错了方法。战略缺陷是指驾驶员的编码实现或者设计最终将不能达到目标。导航员在两人中更多思考长期的战略目标。由于导航员并不深入涉及设计、算法、代码或测试，所以他具有更客观的视角，能够更好地从战略上思考工作方向。结对中的两人是聊天伙伴，会不断进行头脑风暴[75]。有效的结对伙伴会经常讨论解决问题的其他可能方法和方案[67][61]。如果导航员过于安静，那可能是结对运转不良的信号。驾驶员和导航员应该定时更换角色。在一个软件开发团队中，团队成员应该与其他不同的队员结对，以此来更好地利用各自的经验。

“结对编程”这一名字可能会引起人们的误解，认为软件工程师和其他软件开发人员只在编码阶段结对。然而，结对可以发生在开发流程的所有阶段：结对设计、结对调试、结对测试，等等。程序员可以在开发中的任何时刻与人结对，特别是在他们处理复杂和不熟悉的问题时。此外，产品经理可以与用户界面设计师结对，合作开发用户体验。测试人员和开发人员可以结对，合作开发自动化测试。简而言之，结对编程中并肩合作的方式可以在所有开发阶段被许多团队成员使用。

这章会以结对编程的使用简史开始。然后，我将分别提供在产业界和学术界使用结对编程的信息。在这之后，我将讨论分布式结对编程，也就是程序员不在同一个地点时使用结对编程的方法。最后，我将提出大面积推广结对编程所面临的挑战。

17.1　结对编程的历史

“结对编程”在人们给它这个名字之前，已经被提倡并使用了几十年。《人月神话》的作者Fred Brooks[9]说道：“我在读研究生的时候和我的同学Bill Wright首次尝试了结对编程（1953年～1956年）。我们写了1500行没有缺陷的代码，第一次尝试运行就通过了。”[67]

在20世纪80年代早期，曾发表过150多篇技术文章和16部著作的Larry Constantine汇报了在

Whitesmith有限公司观察到的“动态二人组”，他们能更快地产出错误更少的代码[14]。他评论道，两个聪明脑袋合在一起，并相互信任地稳定沟通，能使代码受益匪浅。他也总结说，两个程序员合伙并不多余，相反这是更高效率和更好质量的直达途径。

基于贝尔实验室Pasteur项目的研究结果（一个对50家高效的软件开发组织的大型社会学和人类学研究），James Coplien在1995年发表了“结对开发”的组织模式[15][16]。Coplien把这种模式的力量描述为：“有时人们只有在获得帮助时才感觉他们能解决问题。而有一些问题不是任何一个人单独可以解决的。”这个组织模式提出的解决方案是“让相容的两个设计者结对工作；他们一起的产出多于各自产出的总和”。应用这种模式的结果是“更有效地实现流程，而且两个人结对工作不那么容易出现独自工作时的那些盲区。”

1998年，坦普尔大学的教授John Nosek第一个对结对编程的有效性进行了经验性研究[49]。Nosek汇报了15个有经验的全职程序员的工作，给他们45分钟时间来解决对他们的组织有挑战性的问题。在这些程序员自己的环境和自己的设备中，5个人独自工作，另外10个人结成5对工作。工作条件和工作材料对试验小组（结对）和对照小组（个人）都相同。双侧t分布检验显示，此研究提供了显著的统计结果。结对的小组在任务上总共多花了60%的时间。然而，由于他们是双人作战，他们完成任务的速度比对照小组快了20%。Nosek也汇报了结对小组产出了更好的算法和代码。

正如我之前提到的，在20世纪末和21世纪初，XP软件开发方法论的出现把结对编程推到了前线。XP出现后，许多人对结对编程有所怀疑，因为他们相信如果两个程序员一起工作，花在编码任务上的工作量会翻倍。

在1999年，针对这种怀疑，犹他大学进行了大量的实证研究[65][66][69]，期待找出结对编程的成本及收益。41个软件工程本科三四年级的学生参与了一学期15周的精心设计的实验。上课的第一天，学生被询问是否更愿意结对或单独工作，想与谁一起工作，不想与谁一起工作。学生们也根据他们学习成绩的平均等级分（GPA）被归类为“优等生”（25%最好的学生）、“普通生”和“差等生”（25%最差的学生）。28位学生被分在结对小组，13位被分在单独小组。我们用GPA来保证两组的学术水平相同。14对学生中，13对是互相选择的，也就是每个学生都在之前要求和他的搭档合作。最后一组是分配的，因为他们没有表示对搭档的偏好。

所有的学生都收到了一份有效结对编程的说明，并得到了一篇关于如何才能成功地进行合作的论文[66]，以帮助他们更好地准备。研究使用明确的度量标准来保证结对小组每周都一直工作在一起。每周有一课时安排学生研究他们自己的项目。另外，我们要求学生每周必须与搭档在办公时间合作进行项目两小时。在这些有规律的见面时间中，这些配对逐渐成形而且相互的关系也更紧密，因而也就更有可能安排额外的时间一起来完成他们的工作。

结对小组通过了明显更多的后开发（post-development）自动化测试用例，这些用例是由助教来公平运行的（见表17-1）。结对中的学生平均多通过了15%的教师布置的测试用例。质量差别的统计显著性也达到$p<0.01$。

表17-1　犹他大学结对编程比较

	单独工作学生的测试通过率	结对工作学生的测试通过率
程序1	73.4%	86.4%
程序2	78.1%	88.6%
程序3	70.4%	87.1%
程序4	78.1%	94.4%

研究结果也显示，结对小组平均比独立小组多花了15%的编程时间来完成项目。比如，如果一个人花了10小时在作业上，那结对中的每个个体大约就会花平均5小时45分钟。但是两组所花的时间中值基本相同，平均差没有统计显著性。即使多花了15%的时间，从提升的质量来看，结对编程仍然是经济可行的软件开发实践[20]。犹他大学的研究在有产业经验的研究生中也运行了一个过16周多的学期。然而，这些实验结果只对学术环境适用。

犹他大学的研究结果显示了结对编程在不加倍人力资源的情况下仍然能够获益，在XP实践的运用和犹他大学研究成果基础上，结对编程开始在产业和教育中更流行起来。下两节将概述结对编程在这两种环境下的使用情况。

17.2　产业环境中的结对编程

这一节概述一些在企业中得到证明的结对编程实践以及它们的结果。

17.2.1　结对编程的行业实践

软件行业中的团队汇报了对于结对编程持续稳定的使用经验。当从单独开发转为结对开发时，开发者通常需要几天（大约8～12小时）来熟悉和适应这种动态实践[61]。在上一章所引用的犹他大学的研究中，学生们在第一个程序后就适应了结对编程，大约花了8～10小时。通常，程序员不会整天结对，可持续结对工作的时间长度是1.5～4小时[62]。延长结对编程的时间对开发者来说会很困难，结对工作的快节奏和对手头工作的持续聚焦会使你精疲力竭[67] [62]。

在产业团队中，轮换结对是一种保持动态结对的常见实践，而不是分配完搭档后一次结对几天或几星期。许多团队每天轮换配对好几次[6]，或至少每天一次[24]。经常地轮换结对对团队中的知识传递很有好处[61]。另外，轮换结对可以帮助教导和培训新的团队成员，Menlo Innovations、微软、摩托罗拉、Silver Platter软件以及一家不愿透露姓名的大型意大利制造商已经用经验证明了这一点（详情参见参考文献[6]、[25]、[36]、[73]）。决定与谁结对通常比较随意，也不会有什么困难[12] [61]。经常在一个简短的用来讨论每日情况和当前挑战的例会中就能决定，比如“站立会议”或Scrum中的每日例会（Scrum是一种敏捷软件开发方法论）[54]。

轮换驾驶员和导航员的角色对保持两个软件工程师同时投身工作很重要[61] [67]。然而，在一个为期4个月的研究中，Chong和Hurlbutt观察了两个实践结对开发的专业软件开发团队，他们的结对伙伴中并没有驾驶员和导航员的角色[12]。当程序员能力相同时，他们会共同讨论并想办法。

在这种情况下，驾驶员主要扮演打字者的角色。当程序员能力水平不同时，能力较强的那个软件工程师会主导两人的互动。

Chong和Hurlbutt也注意到了键盘的持有权对决策权有微妙却重要的影响：驾驶员通常是最终决定者。他们注意到键盘通常被来回传递，工程师们在共同做驾驶员和导航员时最有效。当他们得到键盘或感到即将控制键盘时，似乎更投入工作。因此，他们支持使用双份键盘和鼠标。Hofer[31]和Freudenberg等人[23]实施的分析支持Chong和Hurlbutt的发现。

IBM和Guidant的开发团队可以选择使用结对编程或代码检查。得到这个选择权后，一些IBM的团队增加了5%～50%的结对编程使用时间[72]，而Guidant几乎所有时间都在使用结对编程[51]。

通常，软件工程师觉得结对编程应当只应用在编写规范说明、做设计和较为复杂的编程任务上（详情参见参考文献[18]、[32]、[39]、[61]、[62]、[63]）。一个大型的295个咨询师参与的实验证明，在复杂任务上结对提高了质量，而对于简单任务却无任何提高[1]。

团队在使用有组织有结构的结对编程方法时，成功率更高。比如，公开显示结对的小时数，或者在每日例会中分配结对对象，而不是自愿或非正式的使用结对编程。在一个组织中[63]，一项调查显示了对结对编程的正面反馈和更多的使用意愿。然而，这些工程师认为无法使用更多结对编程的原因来源于组织，比如找不到共同的时间、团队负责人不支持、在项目计划时没有考虑结对编程等。

在大五人格因素中有不同的人格类型[64]已被证明会对专业结对编程者的沟通强度有积极的影响[29]。（更多大五人格因素的信息，参见第6章。）然而微软的软件工程师认为个性间的冲撞会破坏结对的生产力[5]，结对双方在有互补技能时工作效果最好。

最后，结对编程者觉得有更大的桌子、更宽的屏幕、无线鼠标和无线键盘以及墙上的白板，这些都对工作有益。结对的噪声会影响其他单独工作的人。专门在旁边为结对编程设置一间屋子或者一块区域可以帮助减轻这个影响[63]。

17.2.2 业内使用结对编程的效果

那些持续使用结对编程的团队让我们对于结对编程在业内的实际使用效果有了更深入的了解。许多团队汇报了使用结对编程后的产品质量提升（详情参见参考文献[5]、[18]、[34]、[39]、[61]）。有一家大型的芬兰电信公司，他们的软件工程师几乎只工作在结对环境中，在一年半的生产过程中只有5个线上故障[61]，可惜并没有基线比较数据存在。另一个芬兰的对照案例分析[32]表明，在某一个项目中，结对和单独开发有着相同的缺陷密度，而在另一个项目中结对开发的缺陷密度是单独开发人员的六分之一。

由于结对编程所产生的知识共享[5] [39]和开发环境改善等积极效应[5] [61] [62]，团队可能因此被激发而建立结对编程制度。如果只有一个人理解一处代码，团队就会因为这个人离开团队、病假或者休假而遇到麻烦。当对每处代码都有不只一个人比较熟悉的时候，整个团队的风险就降低了。

团队发现结对写出的代码更容易理解[61] [62]。驾驶员写出的代码必须让导航员读懂，这样就能激励驾驶员写得更清晰明白。同样，如果驾驶员不理解代码，导航员会让驾驶员停下来澄清，那样驾驶员重写代码时就会写得更简单易懂。

当工程师们刚开始结对编程的时候，有时会有一些生产力的下降[1] [49]。当一个团队持续使用结对编程后，在软件工程师工作在复杂任务上时，结对编程会对生产力产生正面影响[61]。他们猜测其他的研究中所显示的生产力下降[48] [49] [65]是因为只研究了个人的任务或者较小的项目。另一个团队认为他们有一小幅度的生产力下降[61]，这与之前研究所观察到的一样[48] [49] [65]。对四个芬兰团队的对照案例分析[32]并没有显示出任何单个或结对程序员的更高或更低生产力模式。总体说来，这些研究只用了“每人月的代码行数”作为生产力度量，这种度量通常被认为是不精确的。

团队认为使用结对编程提高了团队的士气。他们也认为使用结对编程帮助了他们使用其他实践的纪律性，如测试驱动型开发、代码规范的使用以及频繁集成[61] [62]。

一个涉及8个专业软件开发人员的结构化实验在泰国展开，以比较Fagan检查[21]和结对编程的结果。结对多花了4%的时间完成产品开发，但是在用户验收测试时检测到的缺陷却少了39%[52]。

17.3　教育环境中的结对编程

在大部分情况下，学生们和从业者们用同样的方式结对编程。在这一节中，我们讨论一些学生特有的结对编程实践，并从教育文献中引用一些结对编程的使用结果。

17.3.1　教学中特有的实践

教职员工可能会允许学生选择他们的搭档[13] [33]，也可能主动安排学生配对，让看起来能一起合作得较好的学生结对。那些选择配对的人会使用探索式的方法来慢慢组成最有效的配对。一项研究表明男女搭配能产生更高质量和更有创造性的结果[44]。一项58个本科生参与的研究表明，结对双方如果为他们的思想开放程度和负责程度打出相似的分数时，工作效果最好[11]。另一项54个本科生参与的研究表明，基于大五人格因素模型，责任心和作业得分之间有正相关性，接纳新事物的开放程度和测试成绩之间也有正相关性[53]。

研究人员开展了一项涉及1350个学生的大型实证研究，来调查教职员工预先帮学生协调配对时可以利用的因素[74]。研究显示，大多数情况下（93%的结对学生如此认为）学生认为与他们的搭档协调一致。研究结果也显示，教职员工可以使用以下一种或几种标准来使学生配对更相容。

- 让有相似技能水平的学生结对，可以用计算机科学的GPA和/或总GPA来度量。（Toll等人[60]，Grant Braught等人[26]，以及Sennett和Sherriff也支持配对的学生有相似的技能水平[55]。）
- 让Myers-Briggs个性类型①中的感觉型人和直觉型人配对。
- 让有相似工作规范的学生配对。可以让学生在1～9中选一个数字，1表示做到足以混过去就行了，9表示努力做到最好。选择相近数字的学生配对更好。

轮换结对在教学环境中比产业环境中做得少。通常搭档们在整个作业阶段都在一起工作（1~3周）[58]。一些教育者更喜欢让搭档们持续一整个学期[42]。

① Myers-Briggs个性类型指数（MBTI）是一种著名的人格测试。它把人格分为四对相反的因素，这样所有的人格就被分为16种类型。感觉型和直觉型是MBTI中的一对相反因素。——译者注

17.3.2 教学中使用结对编程的效果

有研究证明，结对编程创造了一个有益于主动学习和社会交往的环境，使学生少受挫，更自信[53]，而且对IT更有兴趣（详情参见参考文献[7]、[37]、[38]、[45]、[57]）。这与传统单独编程教学的缺点形成了强烈对比，传统教学使学生感觉孤立、受挫、对自己的能力不确定。结对编程鼓励学生与班上和实验室的同学多交流，因此创造了更公共、有更多支持的环境。当前一代学生把协作的环境看得很重[50]。另外，结对编程中固有的协作使学生不得不接触产业环境中所需的协作、团队合作、交流技巧，并使他们从中得到锻炼。这种益处似乎能帮助计算机科学留住人才，特别是女性（详情参见参考文献[10]、[41]、[42]、[71]）。总的说来，结对编程提供了一种真实反映出“实验室模型”的方法，这在自然科学领域（如物理和化学）是一种常用的实践[68]。

结对工作的学生通常产出更高质量的项目，并有更高的课程通过率（详情参见参考文献[42]、[71]、[8]、[76]、[43]），即使当学生在分布式环境下结对编程结果也如此（详见下一节）[28]。在入门级课程中结对的团队在将来需要单独编程的课程中更成功[71] [33]。一项在佩斯大学本科生中进行的研究发现，根据学生开发项目和考试分数的成绩来看，课外协作与学生表现有正相关性[35]。在一项迪金森学院（一所文科院校）涉及本科生的研究中，即使是SAT分数较低的学生在使用结对编程后也能达到更高的实验分数[8]。

结对编程也能使教职员工受益。由于只有一半的作业提交，评分的工作就少了许多。结对的学生通常能解决低级的技术或程序问题，而这些问题往往会占据实验室助教和教师大量时间，也是其邮件中的主要主题[29] [70]。最后，“问题学生”也会相应减少，因为结对编程中来自同伴的压力鼓励所有学生积极参与课程。学生不希望影响同伴的成绩，因此会对布置的作业更努力，通常比单独工作的情况下开始得更早（虽然并不是所有学生都开始得更早[56]）。

固然，实施结对编程也会有一些成本。对学生来说，主要有两项成本没有明显的解决方法。一小部分学生（大约5%）总需要独自工作[74]。这些常常是尖子学生，他们不希望被其他学生拖后腿，也不觉得指导别人对他们有益。另一个问题是在课程和实验室之外结对编程时，需要协调时间安排。

17.4 分布式结对编程

分布式软件开发在产业界已经是常用的实践了。在教学中，学生也更喜欢在他们的寝室或家里工作，而不是去实验室和他们的搭档工作。另外，登记远程教学课程的学生可能永远不能面对面交流。分布式的工作者可以使用一系列工具通过因特网实践结对编程。在最简单的情况下，程序员可以使用VNC①或Windows会议空间②（之前的网络会议）来共享桌面。这些工具能把一方任何应用程序所显示的输出传送给其他人，这当然需要个人之间足够的带宽、信任和安全性。其他工具，如Sangam[30]、xpairtise③、COPPER[46]或Facetop[47]只传播对结对编程重要的信息，比如驾

① http://www.realvnc.com/。

② http://www.microsoft.com/windows/products/windowsvista/features/details/meetingspace.mspx。

③ http://xpairtise.sourceforge.net/。

驶员最新的改动。

分布式认知专家Nick Flor强调了分布式结对编程系统对支持跨工作空间的视觉、手工、音频通道的重要性[22]。这些通道使结对双方相互合作，为进行中的知识共享和互助行为提供了微妙却重要的催化剂。比如，微妙的手势，如摇头或喃喃自语对结对间的信息互换是一种催化剂。通过Facetop[47]展示在屏幕上的清晰的搭档形象可以帮助这些信息通道的传播。此外，Chong和Hurlbutt[12]不鼓励使用定义了驾驶员和导航员角色的工具，如Sangam[30][17]。因为它们约束了更有效的结对编程行为：在整个时间段中同时扮演驾驶员和导航员的角色。

在北卡罗来纳州立大学和北卡罗来纳大学教堂山分校，已经进行了一些分布式结对编程的研究[2][3]。这些研究表明，通过因特网结对比分布式非结对团队更有潜力，非结对团队的程序员独立工作，在更晚的时候集成代码。在这些研究中，学生使用桌面共享软件NetMeeting，以及雅虎通、耳机和麦克风来进行交流。也有研究者对印第安纳大学伯明顿分校的信息学本科生进行了一项关于计算机科学入门课程的调查，学生使用Adobe Connect Now网络会议软件进行分布式结对编程[19]。结果表明，学生普遍看好结对编程，但是对分布式结对编程却远没有那么热衷。

17.5　面对的挑战

之前讨论的结果显示了团队使用结对编程实践后能从各种角度获益。然而一些挑战阻止了结对编程的传播。在这一节中，我们列出四个挑战，并对如何克服挑战提出建议。

- 偏好和习惯

  软件工程师习惯于独立工作，特别是那些没有受过结对编程实践相关教育的。因此，许多人担心他们与其他人一起工作的时候不能专心，也许他们会和速度慢的程序员一起“浪费时间”，也许他们和其他伙伴比起来能力不足，也许还有其他的担心。然而，在那些试过此实践的程序员中，他们中的90%更喜欢结对编程而不是独立编程[69][59]。因为大部分工程师最终会喜欢这个实践，所以克服这个挑战的策略是以非威胁性的小范围试点开始。这些工程师很有可能会发现结对编程的益处，所以就更可能会在遇到有挑战的编程任务时和同事们自发开始结对编程。

- 经济利益

  虽然研究证明了相反的结果，但是软件公司仍可能会担心结对编程会使软件开发的成本翻倍，因为两个程序员工作在一个任务上[5]。一个微软老板的话曾经被引用，“如果我能选择的话，我会雇用一个星级程序员而不是两个需要结对的程序员。”[5]管理层应该受到结对编程的研究和使用经验的启发。可以从小范围开始实践，那只会有很少的经济风险。然后，管理层就可以得到一手的理解：这个实践不会引发产品生命周期成本的提高，特别在考虑到提升质量所带来的利益的情况下。

- 协调时间

  当团队中的工程师实践结对编程时，团队必须决定每天谁和谁工作，也必须协调时间[5]。此外，团队也需要选择一天中的哪几个小时是用来结对的，在这些时间中，不能有会议与之

冲突。决定结对组成的理想场合是每日10到15分钟的Scrum例会[54]。在Scrum会议中，工程师们谈论他们遇到的障碍和他们每天将涉及的任务。在这个会议上，可以基于当日的工作任务和当前的挑战动态配对。

- 分布式团队

一些团队可能感觉他们不能结对编程，因为他们的团队成员不在同一地点工作。正如前文所讨论的，分布式结对编程被证明是一种可行的有益实践。

17.6 经验教训

使用过结对编程的产业团队已经意识到了许多好处，包括更高的产品质量、更短的交付时间、相互学习的促进、由于更好的知识管理而减少的产品风险，以及团队精神的提高。尽管有这些益处，产业界整体的结对编程使用率并不高，主要是因为前一节中所讨论的那些原因。一小部分公司是公认的结对编程“商铺”，去那里面试的员工都了解这个实践。在许多其他公司中，结对编程通常在团队成员面临挑战或复杂任务，或者作为新员工培训的一部分时才被主动实施。

学生们和产业团队意识到了同样的好处，但是研究结果显示了对于学术界的额外好处：保留信息技术领域的人才，减轻完成课堂作业的挫败感，让学生更多地认识他们的同学以增加满足感。世界上许多导师也已经意识到这样教学的好处：由于评分工作的减少和回答所有学生技术问题的减少，结对编程能减轻他们自己的工作量。因此，结对编程会在教学中越来越普遍。毕业了的学生会把结对编程看成是更自然的实践。由此，在未来几年，结对编程在产业界的使用率非常可能会有所提升。

17.7 致谢

这里的一些材料基于美国国家科学基金会授予号ITWF 00305917和BPC 0540523所支持的工作。这些材料中的任何意见、发现、结论或推荐都是来自作者的，不一定代表国家科学基金会的观点。

17.8 参考文献

[1] [Arisholm et al. 2007] Arisholm, E., H. Gallis, T. Dybå, and D. Sjøberg. 2007. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Transactions in Software Engineering* 33(2): 65-86.

[2] [Baheti et al. 2002a] Baheti, P., E. Gehringer, and D. Stotts. 2002. Exploring the Efficacy of Distributed Pair Programming. In *Lecture Notes in Computer Science, volume 2418: Extreme Programming and Agile Methods XP/Agile Universe 2002*, ed. D. Wells and L. Williams, 208-220. Berlin: Springer-Verlag.

[3] [Baheti et al. 2002b] Baheti, P., L. Williams, E. Gehringer, and D. Stotts. 2002. Exploring Pair Programming in Distributed Object-Oriented Team Projects. Paper presented at the 11th OOPSLA Educators' Symposium, November 4-8, in Seattle, WA.

[4] [Beck 2005] Beck, K. 2005. *Extreme Programming Explained: Embrace Change*, Second Edition. Reading, MA: Addison-Wesley.

[5] [Begel and Nagappan 2008] Begel, A., and N. Nagappan. 2008. Pair programming: What's in it for me? *Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement*: 120-128.

[6] [Belshee 2005] Belshee, A. 2005. Promiscuous pairing and beginner's mind: embrace inexperience. *Proceedings of the Agile Development Conference*: 125-131.

[7] [Berenson et al. 2005] Berenson, S.B., L. Williams, and K.M. Slaten. 2005. Using Pair Programming and Agile Development Methods in a University Software Engineering Course to Develop a Model of Social Interactions. Paper presented at Crossing Cultures, Changing Lives Conference, July 31-August 3, in Oxford, UK.

[8] [Braught et al. 2008] Braught, G., L.M. Eby, and T. Wahls. 2008. The effects of pairprogramming on individual programming skill. *Proceedings of the 39th SIGCSE technical symposium on computer science education*: 200-204.

[9] [Brooks 1975] Brooks, F.P. 1975. *Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley.

[10] [Carver et al. 2007] Carver, J., L. Henderson, L. He, J. Hodges, and D. Reese. 2007. Increased Retention of Early Computer Science and Software Engineering Students Using Pair Programming. *Proceedings of the 20th Conference on Software Engineering Education and Training*: 115-122.

[11] [Chao and Atli 2006] Chao, J., and G. Atli. 2006. Critical personality traits in successful pair programming. *Proceedings of the conference on AGILE 2006*: 89-93.

[12] [Chong and Hurlbutt 2007] Chong, J and T. Hurlbutt. 2007. The Social Dynamics of Pair Programming. *Proceedings of the 29th International Conference on Software Engineering*: 354-363.

[13] [Cicirello 2009] Cicirello, V.A. 2009. On self-selected pairing in CS1: who pairs with whom? *Journal of Computing Sciences in Colleges* 24(6): 43-49.

[14] [Constantine 1995] Constantine, L.L. 1995. *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press.

[15] [Coplien 1995] Coplien, J.O. 1995. A Development Process Generative Pattern Language. In *Pattern Languages of Program Design*, ed. James O. Coplien and Douglas C. Schmidt, 183-237. Reading, MA: Addison-Wesley.

[16] [Coplien and Harrison 2005] Coplien, J.O., and N.B. Harrison. 2005. *Organizational Patterns of Agile Software Development.* Upper Saddle River, NJ: Pearson Prentice-Hall.

[17] [Devide et al. 2008] Devide, J.V.S., A. Meneely, C.-W. Ho, L. Williams, and M. Devetsikiotis. 2008. Jazz Sangam: A Real-Time Tool for Distributed Pair Programming on a Team Development Platform. Paper presented at the First International Workshop on Infrastructure for Research in Collaborative Software Engineering, November 9, in Atlanta, GA.

[18] [Dybå et al. 2007] Dybå, T., E. Arisholm, D. Sjøberg, J. Hannay, and F. Shull. 2007. Are Two Heads Better Than One? On the Effectiveness of Pair Programming. *IEEE Software* 24(6): 12-15.

[19] [Edwards et al. 2010] Edwards, R.L., J.K. Stewart, and M. Ferati. 2010. Assessing the effectiveness of distributed pair programming for an online informatics curriculum. *Inroads* 1(1): 48-54.

[20] [Erdogmus and Williams 2003] Erdogmus, H., and L. Williams. 2003. The Economics of Software Development by Pair Programmers. *The Engineering Economist* 48(4): 283-319.

[21] [Fagan 1986] Fagan, M.E. 1986. Advances in Software Inspection. *IEEE Transactions on Software Engineering* 12(7): 744-751.

[22] [Flor 2006] Flor, N. 2006. Globally Distributed Software Development and Pair Programming. *Communications of the ACM* 49(10): 57-58.

[23] [Freudenberg et al. 2007] Freudenberg, S., P. Romero, and B. du Boulay. 2007. "Talking the talk": Is intermediate-level conversation the key to the pair programming success story? *Proceedings of AGILE 2007*: 84-91.

[24] [Frever and Ingalls 2006] Frever, T., and P. Ingalls. 2006. The pairing session as the atomic unit of work. *Proceedings of the Conference on AGILE 2006*: 165-169.

[25] [Fronza et al. 2009] Fronza, I., A. Sillitti, and G. Succi. 2009. An interpretation of the results of the analysis of pair programming during novices integration in a team. *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*: 225-235.

[26] [Grant Braught and Wahls 2010] Grant Braught, J.M. and Tim Wahls. 2010. The benefits of pairing by ability. *Proceedings of the 41st ACM technical symposium on computer science education*: 249-253.

[27] [Hanks 2005] Hanks, B. 2005. Student Performance in CS1 with Distributed Pair Programming. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*: 316-320.

[28] [Hanks 2007] Hanks, B. 2007. Problems Encountered by Novice Pair Programmers. *Proceedings of the Third International Workshop on Computing Education Research*: 159-164.

[29] [Hannay et al. 2010] Hannay, J.E., E. Arisholm, H. Engvik, and D.I.K. Sjøberg. 2010. Effects of Personality on Pair Programming. *IEEE Transactions on Software Engineering* 36(1): 61-80.

[30] [Ho et al. 2004] Ho, C-w., S. Raha, E. Gehringer, and L. Williams. 2004. Sangam: A Distributed Pair Programming Plug-in for Eclipse. *Proceedings of the 2004 OOPSLA workshop on Eclipse technology eXchange*: 73-77.

[31] [Höfer 2008] Höfer, A. 2008. Video Analysis of Pair Programming. *Proceedings of the 2008 International Workshop on Scrutinizing Agile Practices*:37-41.

[32] [Hulkko and Abrahamsson 2005] Hulkko, H., and P. Abrahamsson. 2005. A Multiple Case Study on the Impact of Pair Programming on Product Quality. *Proceedings of the 27th InternationalConference on Software Engineering*: 495-504.

[33] [Jacobson and Schaefer 2008] Jacobson, N., and S.K. Schaefer. 2008. Pair programming in CS1: overcoming objections to its adoption. *SIGCSE Bulletin* 40(2): 93-96.

[34] [Jensen 2003] Jensen, R. 2003. A Pair Programming Experience. *CrossTalk* 16(3): 22-24.

[35] [Joseph and Payne 2003] Joseph, A., and M. Payne. 2003. Group Dynamics and Collaborative Group Performance. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*: 368-371.

[36] [Lacey 2006] Lacey, M. 2006. Adventures in Promiscuous Pairing: Seeking Beginner's Mind. *Proceedings of the conference on AGILE 2006*: 263-269.

[37] Layman 2006] Layman, L. 2006. Changing Students' Perceptions: An Analysis of the Supplementary Benefits of Collaborative Software Development. *Proceedings of the 19th Conference on Software Engineering Education and Training*: 156-166.

[38] [Layman et al. 2005] Layman, L., L. Williams, J. Osborne, S. Berenson, K. Slaten, and M. Vouk. 2005. How and Why Collaborative Software Development Impacts the Software Engineering Course. *Proceedings of the 35th Annual Conference on Frontiers in Education*: T4C 9-14.

[39] [Luck 2004] Luck, G. 2004. Subclassing XP: Breaking its rules the right way. *Proceedings of the Agile Development Conference 2004*: 114-119.

[40] [McDowell et al. 2002] McDowell, C., L. Werner, H. Bullock, and J. Fernald. 2002. The Effects of Pair Programming on Performance in an Introductory Programming Course. *Proceedings of the 33rd SIGCSE technical symposium on computer science education*: 38-42.

[41] [McDowell et al. 2003] McDowell, C., L. Werner, H. Bullock, and J. Fernald. 2003. The Impact of Pair Programming on Student Performance, Perception, and Persistence. *Proceedings of the 25th International Conference on Software Engineering*: 602-607.

[42] [McDowell et al. 2006] McDowell, C., L. Werner, H. Bullock, and J. Fernald. 2006. Pair Programming Improves Student Retention, Confidence, and Program Quality. *Communications of the ACM* 49(8): 90-95.

[43] [Mendes et al. 2006] Mendes, E., L. Al-Fakhri, and A. Luxton-Reilly. 2006. A Replicated Experiment of Pair Programming in a 2nd Year Software Development and Design Computer Science Course. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*: 108-112.

[44] [Mujeeb-u-Rehman et al. 2005] Mujeeb-u-Rehman, M., X. Yang, J. Dong, and M. Abdul Ghafoor. 2005. Heterogeneous and homogenous pairs in pair programming: an empirical analysis. *Proceedings of the Canadian Conference on Electrical and Computer Engineering 2005*: 1116-1119.

[45] [Nagappan et al. 2003] Nagappan, N., L. Williams, M. Ferzli, K. Yang, E. Wiebe, C. Miller, and S. Balik. 2003. Improving the CS1 Experience with Pair Programming. *Proceedings of the 34th SIGCSE technical symposium on computer science education*: 359-362.

[46] [Natsu et al. 2003] Natsu, H., J. Favela, A. Morán, D. Decouchant, and A. Martinez-Enriquez. 2003. Distributed Pair Programming on the Web. *Proceedings of the 4th Mexican International Conference on Computer Science*: 81-88.

[47] [Navoraphan et al. 2006] Navoraphan, K., E. F. Gehringer, J. Culp, K. Gyllstrom, and D. Stotts. 2006. Next-generation DPP with Sangam and Facetop. Proceedings of the 2006 *OOPSLA workshop on Eclipse technology eXchange*: 6-10.

[48] [Nawrocki and Wojciechowski 2001] Nawrocki, J., and A. Wojciechowski. 2001. Experimental Evaluation of Pair Programming. *Proceedings of the 12th European Software Control and Metrics Conference*: 269-276.

[49] [Nosek 1998] Nosek, J.T. 1998. The Case for Collaborative Programming. *Communications of the ACM* 41(3): 105-108.

[50] [Oblinger 2003] Oblinger, D. 2003. Boomers, Gen-Xers, and Millennials: Understanding the New Students. *Educause Review* 38(4): 37-47.

[51] [Pandey et al. 2003] Pandey, A., C. Miklos, M. Paul, N. Kameli, F. Boudigou, V. Vijay, A. Eapen, I. Sutedjo, and W. Mcdermott. 2003. Application of tightly coupled engineering team for development of test automation software—a real world experience. *Proceedings of the 27th Annual International Computer Software and Applications Conference*: 56-63.

[52] [Phongpaibul and Boehm 2006] Phongpaibul, M., and B. Boehm. 2006. An Empirical Comparison Between Pair Development and Software Inspection in Thailand. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*: 85-94.

[53] [Salleh et al. 2009] Salleh, N., E. Mendes, J. Grundy, and G.S.J. Burch. 2009. An empirical study of the effects of personality in pair programming using the five-factor model. *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*: 214-225.

[54] [Schwaber and Beedle 2002] Schwaber, K., and M. Beedle. 2002. *Agile Software Development with SCRUM*. Upper Saddle River, NJ: Prentice-Hall.

[55] [Sennett and Sherriff 2010] Sennett, J., and M. Sherriff. 2010. Compatibility of Partnered Students in Computer Science Education. *41st ACM Technical Symposium on Computer Science Education* (*SIGCSE*): 244-248.

[56] [Simon and Hanks 2007] Simon, B., and B. Hanks. 2007. First Year Students' Impressions of Pair Programming in CS1. *Proceedings of the Third International Workshop on Computing Education Research*: 73-86.

[57] [Slaten et al. 2005] Slaten, K.M., M. Droujkova, S. Berenson, L. Williams, and L. Layman. 2005. Undergraduate Student Perceptions of Pair Programming and Agile Software Methodologies: Verifying a Model of Social Interaction. *Proceedings of the Agile Conference 2005*: 323-330.

[58] [Srikanth et al. 2004] Srikanth, H., L. Williams, E. Wiebe, C. Miller, and S. Balik. 2004. On Pair Rotation in the Computer Science Course. *Proceedings of the 17th Conference on Software Engineering Education and Training*: 144-149.

[59] [Succi et al. 2002] Succi, G., M. Marchesi, W. Pedrycz, and L. Williams. 2002. Preliminary analysis of the effects of pair programming on job satisfaction. *Proceedings of the Fourth International Conference on eXtreme Programming and Agile Processes in Software*: 212-215.

[60] [Van Toll et al. 2007] Van Toll, T., III, T., R. Lee, and T. Ahlswede. 2007. Evaluating the Usefulness of Pair Programming in a Classroom Setting. *Proceedings of the 6th IEEE/ACIS International Conference on Computer and Information Science* (*ICIS*) *2007*: 302-308.

[61] [Vanhanen and Korpi 2007] Vanhanen, J., and H. Korpi. 2007. Experiences of Using Pair Programming in an Agile Project. *Proceedings of the 40th Annual Hawaii International Conference on System Sciences* (*HICSS*) *2007*: 274b.

[62] [Vanhanen and Lassenius 2007] Vanhanen, J., and C. Lassenius. 2007. Perceived Effects of Pair Programming in an Industrial Context. *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*: 211-218.

[63] [Vanhanen et al. 2007] Vanhanen, J., C. Lassenius, and M. Mäntylä. 2007. Issues and Tactics when Adopting Pair Programming: A Longitudinal Case Study. *Proceedings of the International Conference on Software Engineering Advances* (*ICSEA*) *2007*: 70.

[64] [Walle and Hannay 2009] Walle, T., and J.E. Hannay. 2009. Personality and the nature of collaboration in pair programming. *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*: 203-213.

[65] [Williams 2000] Williams, L. A. 2000. The Collaborative Software Process. PhD diss., University of Utah.

[66] [Williams and Kessler 2000] Williams, L. A. and R. R. Kessler. 2000. All I Ever Needed to Know About Pair Programming I Learned in Kindergarten. *Communications of the ACM*. 43(5): 108-114.

[67] [Williams and Kessler 2003] Williams, L., and R. Kessler. 2003. *Pair Programming Illuminated*. Reading, MA: Addison-Wesley.

[68] [Williams and Layman 2007] Williams, L,. and L. Layman. 2007. Lab Partners: If They're Good Enough for the Natural Sciences, Why Aren't They Good Enough for Us? *Proceedings of the 20th Conference on Software Engineering Education and Training*: 72-82.

[69] [Williams et al. 2000] Williams, L., R. Kessler, W. Cunningham, and R. Jeffries. 2000. Strengthening the Case for Pair-Programming. *IEEE Software* 17(4):19-25.

[70] [Williams et al. 2002] Williams, L., E. Wiebe, K. Yang, M. Ferzli, and C. Miller. 2002. In Support of Pair Programming in the Introductory Computer Science Course. *Computer Science Education* 12(3):197-212.

[71] [Williams et al. 2003] Williams, L., C. McDowell, N. Nagappan, J. Fernald, and L. Werner. 2003. Building Pair Programming Knowledge Through a Family of Experiments. *Proceedings of the 2003 International Symposium on Empirical Software Engineering*: 143-152.

[72] [Williams et al. 2004a] Williams, L., W. Krebs, L. Layman, A. Antón, and P. Abrahamsson. 2004. Toward a Framework for Evaluating Extreme Programming. *Proceedings of Empirical Assessment in Software Eng.* (*EASE*) *2004*: 11-20.

[73] [Williams et al. 2004b] Williams, L., A. Shukla, and A. Antón. 2004. An Initial Exploration of the Relationship Between Pair Programming and Brooks' Law. *Proceedings of the Agile Development Conference 2004*: 11-20.

[74] [Williams et al. 2006] Williams, L., L. Layman, J. Osborne, and N. Katira. 2006. Examining the Compatibility of Student Pair Programmers. *Proceedings of the Conference on Agile 2006*: 411-420.

[75] [Wray 2010] Wray, S. 2010. How Pair Programming Really Works. *IEEE Software* 27(1): 50-55.

[76] [Xu and Rajlich 2005] Xu, S., and V. Rajlich. 2005. Pair Programming in Graduate Software Engineering Course Projects. *Proceedings of the 35th Annual Frontiers in Education Conference*: F1G7-F1G12.

第18章

现代化代码审查

Jason Cohen

18.1　常识

为什么这本书的每一页在出版之前都要由编辑审稿呢？

因为即使你是最聪明、最能干、最有经验的作家，你也不能校对自己的工作。你与书中的那些概念太接近了，那些词汇在你的大脑里面转了太长时间，你已经无法作为第一次读到这些词汇的人来审阅自己的文章了。

写代码也如此。事实上，写文章不可能缺少独立的审稿人，写代码也不可能孤立地写。代码必须在微小的细节上也是正确的，而且，其中也包括写给人看的文字。（你不是也写代码注释吗？）

普通意义上，二个人总是强过一个人，尤其是当审查者是这个领域的专家，或者写代码的人是对这个领域还不熟悉的初级程序员时。

只是，前景并不乐观。事实上，代码审查占用了昂贵的代码开发时间。毕竟，一个九十分钟的四人会议相当于六个小时的工作时间——一个人一天工作的时间。这是相当大的时间投入，所以我们必须要问问这样做的好处是否大于它的支出。

幸运的是，我们有相应的数据。在过去的十年中，我们对于如何有效地执行代码审查有了长足的进步。如果执行正确，代码审查会比测试或者其他调试技术更快发现缺陷；但如果代码审查效率低，会很快变得毫无成效。

本项调查首先会给出各类代码审查都适用的实践，从正规的审查到程序员独自复查自己的代码。讨论完这些基础的最佳实践之后，我们会讨论团队审查中的数据。

18.2　程序员独立进行小量代码审查

代码审查的方法有成千上万种，但有一种是基本的，那就是单独一个程序员严格地审查代码。所以，讨论一下程序员们在专心致志、不与别人讨论的情况下审查自己代码的统计结果是十分有意义的。

当程序员独自审查代码时，哪些方法最有效？

18.2.1　防止注意力疲劳

一次代码审查应该花多少时间呢？

图18-1表示随着时间的推移，发现缺陷效率的变化。横轴表示时间，竖轴表示在该时间点上所发现缺陷的平均数。

很明显，在代码审查的早期，时间与所发现的缺陷数量之间有一个线性关系：（平均）每10分钟就会有一个缺陷被找到。这是鼓舞人心的——它意味着投入代码审查的时间越多，所发现的缺陷就越多。

让我们在这里停下来思考一下。还有其他你所知的软件开发流程能够每10分钟发现（并且通常修改好）一个新的缺陷吗？当然没有，在通常的软件开发/测试环节中，测试人员和开发人员花费的时间远远不止这些。

但在60分钟左右的时候，突然出现一个下降。这个时间点之后，每10分钟不再必然发现另一个缺陷。发现缺陷的效率急剧下降。

通常用来解释这个现象的理论是，我们的大脑大约一小时就会疲倦；在技术工作上集中精力一小时，已经是很长一段时间了。所以不管是什么原因，每次代码审查的时间很显然不应该超过一个小时。

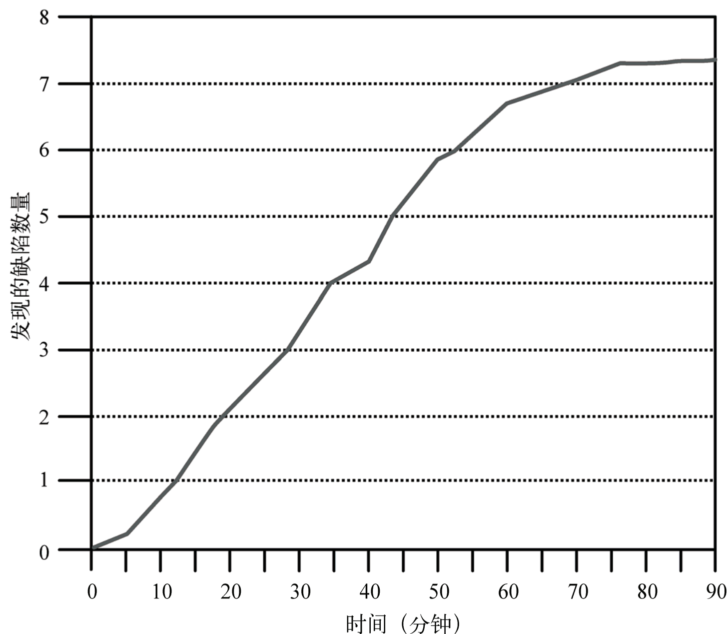


图18-1　60~90分钟后，我们发现错误的能力急降

18.2.2　切忌速度过快

逻辑上来说，你在一段代码上投入的时间越长，代码分析就会越深入和细致，发现的代码缺

陷也会越多。同理，如果代码审查十分仓促，那么只能找到最浅显的错误。

那么，在重要的问题上投入充分的时间追根溯源，而不在完善的代码上浪费时间，这两者之间如何找到一个平衡呢？

图18-2是一项基于2500份代码审查的研究结果，显示了代码审查速度对于发现缺陷能力的影响[1]。

横轴表示审查速度，用每小时审查的代码行数（LOC）表示。竖轴表示缺陷密度，表示每1000行代码（kLOC）发现缺陷的数量。这里用“缺陷密度”而不是“缺陷数量”，因为审查的代码越多，自然会发现越多的缺陷，所以用“缺陷密度”标准化了不同长度代码的审查结果。

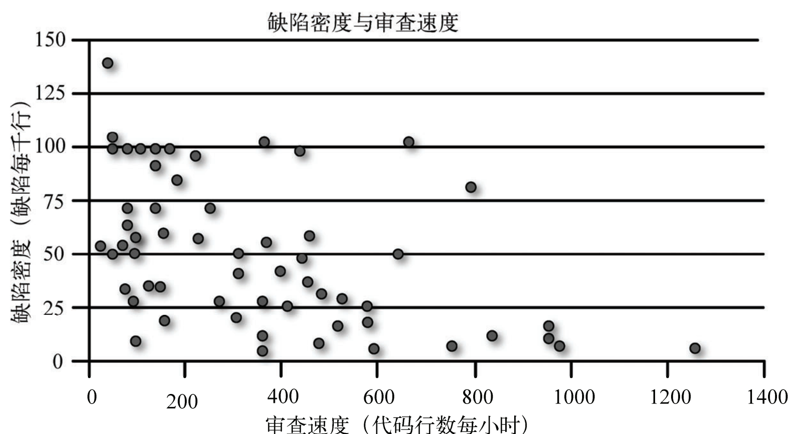


图18-2 在每小时审查400～500行代码的时候，发现缺陷的效率下降

在每小时0行与大约400行代码之间，缺陷密度均匀分布在每千行代码0个和100个缺陷之间。这是符合预期的。比如说，被审查的代码负责往一个接口里添加文件。三百行代码中可能只有很少的缺陷或者没有缺陷，也就是说，缺陷密度很低。但如果被审查的是一段逻辑复杂、多线程、高性能的代码，并且处于核心模块中，整个应用都依赖于这个模块，那情况就不同了。或许只有四行代码被改变了，但由于这个问题的复杂性，必须保证它是正确的，几位审查者对这段代码吹毛求疵后，好不容易挑出了两个缺陷，于是缺陷密度就很高。

简而言之，不同的代码审查所得到的缺陷密度自然会有差异，所以这样的分布是很合理的。

审查速度一旦超过每小时400~500行代码就会有问题。突然间缺陷密度较高的代码审查极少出现。当然不是因为没有缺陷，而是因为审查速度过快以至于发现不了缺陷。

18.2.3 切忌数量过大

综合之前的两个结果，我们的结论是：如果每次审查代码的时间不能超过一小时，并且一个小时之内审查的代码数量不超过400行，那么进行一次代码审查就不能超过400行代码。

让我们来测试一下这个听起来不错的理论。

图18-3显示了代码审查数量对缺陷密度的影响[1]。与上一部分的结果相似，我们预计缺陷密

度分布在每千行代码0~100缺陷。但是当审查的代码数量增加时，缺陷密度降低。正如我们预期的那样，这个时间点大约在300~500LOC时。所以很清楚，那时代码审查的效率不高。

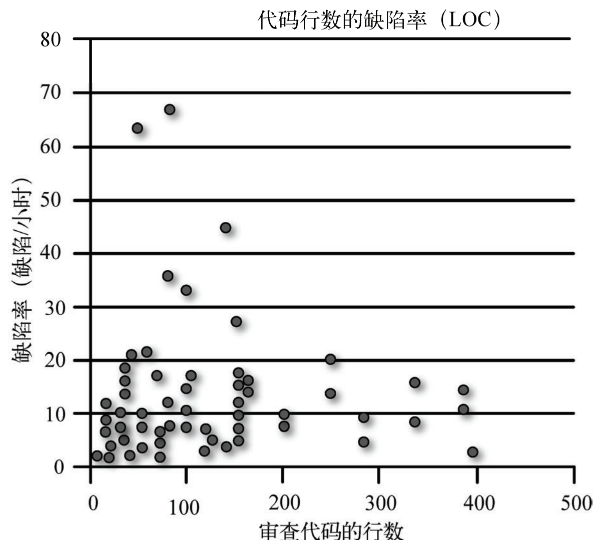


图18-3　当代码数量增加时，我们发现缺陷的能力降低了。所以一次不要审查超过400行代码

实验结果与预期十分吻合，所以三个实验结果都是可信的。

18.2.4　上下文的重要性

常用的版本控制工具Diff只显示改动以及其上下文中的几行。即使是最复杂的工具每次也只显示一个文件。

限制你视野的工具会使你在代码审查中变得近视。毕竟你审查的只可能是摆在你面前的东西！

回头想想，这似乎是一个很糟糕的实践。想想你在审查单个函数内的一个缺陷修复时发生了什么。你审查了改动的代码，发现它修复了原本的问题，但代码审查并不是就到此为止了。接下来的问题是：依赖于这个函数的代码会不会依赖于原来的代码？如果有副作用，还有什么其他代码会受影响？是否有相应的单元测试？其他不同函数的单元测试会不会因为这里的改动而需要修改？这可以接受吗？

因此，与只审查眼前的改动相反，审查代码的改动对于其他文件、类、文档和其他上下文的影响有多重要呢？一项实验中，审查者们掌握了各种不同上下文信息——与当前修改相关的文件[3]。它用了一个简单的自动化工具，基于调用或者被调用的方法，产生了“相关文件”列表，这样多发现了15%的缺陷。当人为来判断相关文件时，可以多发现30%的缺陷。

所以我们建议不要孤立地进行代码审查。至少你要在你的IDE（集成开发环境）前，可以看

到所有的代码库。如果代码的作者可以标出相关代码，那么时间的利用率会提高。

18.3 团队影响

至此，我们已经考虑过单个程序员单独审查代码的情况，但完整的流程从来不止于此。

实际上，代码审查是为数不多程序员共同完成的事情之一。对于代码审查来说，这点本身就很有趣。大家都同意一定程度的交互合作对于传播学术知识，互相教授各种技巧，解决一些棘手的问题是有用的。

但如果说浪费一个人的时间是昂贵的，那么浪费几个人的时间绝对是毁灭性的。所以我们需要看一下相关数据，确保没有把全部工作时间都冲到马桶里面去了。

18.3.1 是否有必要开会

当你想到代码审查的时候，脑海里中也许会浮现出这样一幅画面：会议室中，一群狂人蜷缩在投影仪旁，一边听某个倒霉的程序员解释她的代码，一边批判她。

除了代码作者本身的处境不让人羡慕之外，这样的方式也可能十分浪费时间。Michael Fagan倡导的传统标准审查[4]中，其过程的核心就是一个两个小时的会议，有四个与会者来讨论代码。

四个人两个小时的会议意味着八个人时——相当于一个人一天的时间。其中还不包括事先预定会议室和等待迟到人员的时间。这样做的话最好能替将来省下很多时间，这才能证明这个流程有意义。

在Fagan的理论中会议是必需的，不只是为了方便。他的理论是当四个人在一起的时候，会产生协同效应（原话），团体的力量大于四个人加起来的力量。相当于会议室中有第五个人，一个虚拟的人，能够找出单独一个人找不到的缺陷，Fagan把这第五个人称作“幽灵审查者”。

这是一个听上去很炫的理论，但真的是这样吗？不幸的是，Fagan并没有给出相关数据，17年之后才有相关数据发布[7]。1993年Lawrence Votta统计了代码审查会议之前发现的缺陷数量（当人们自己审查代码时）和会议中发现缺陷的数量。结果如图18-4所示。

96%的缺陷是自己审查代码时发现的，而不是在会议中。回想一下这么说是有道理的，你发现隐匿缺陷的时候，应该是在独自关注代码，而不是与一群人在一起浏览代码，同时还要跟上代码作者对代码的解释。

Fagan理论值得赞扬的是，他发现会议中发现的代码缺陷往往更加隐匿，意味着之后发现和修改它的工作量很大。但是，除了那些最关键的代码之外，程序员亲自参与会议不划算，太浪费时间。

18.3.2 虚假缺陷

虽然会议形式的代码审查从消耗的时间来看并没有发现足够多的缺陷，但是却有另一个作用：它可以修正虚假缺陷，审查者认为是一个问题，事实上却不是。虚假缺陷严重影响效率，因为它需要程序员花时间修正，并可能在这个过程中引入真正的缺陷。

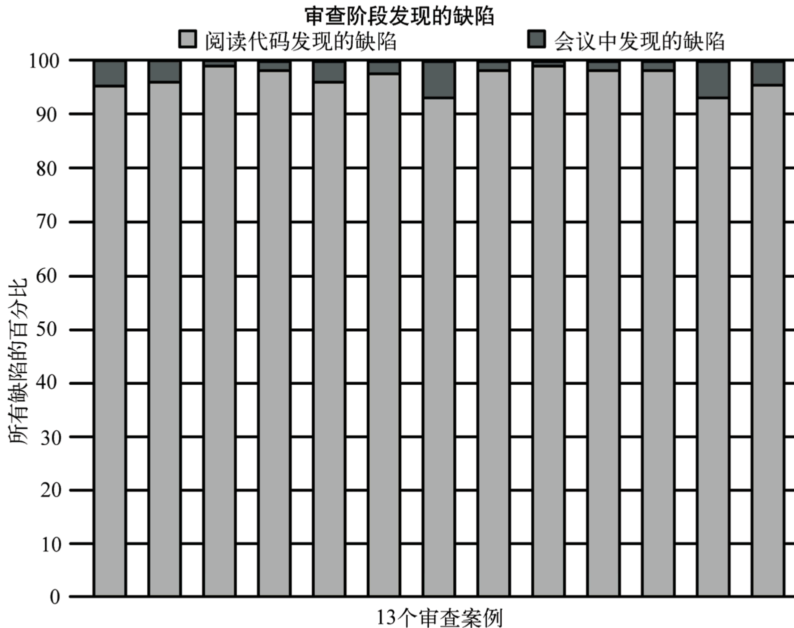


图18-4　Votta展示了代码审查会议额外发现缺陷的数目只占所有被发现缺陷的4%

Votta在他的研究中发现[7]，会议之前认为是缺陷的问题有25%都不是问题，是审查者理解有误。除了Votta，其他一些研究也发现该比率在15%到30%之间[2] [5] [6]。

问题就是修改这些虚假缺陷的代价是什么？这些代价会比会议本身所耗的时间大吗？

一般来说，如果一个人对于一段代码很疑惑，那么其他人很有可能也会疑惑。那么，如果加入一句注释就可以解决这个问题，还是很值得花这个时间的。

毕竟，在同一个房间里开会并不是发现虚假缺陷的唯一方法。只要审查者和代码作者可以通过电话、邮件、或者其他工具进行沟通，他们就可以讨论有疑问的缺陷，大部分虚假缺陷就会消失。关键在于会议形式的审查代码是没有必要的。

18.3.3　外部审查真的需要吗

我们一直假设二个人总是强过一个人，但真的是这样吗？

回到编辑自己作品的比喻上，你自己的确不能看到屏幕上的错误。但如果把它打印出来，在不同的媒介中查看自己的作品，往往问题就会跳出来。或者，一星期之后，再审查自己的作品，你可以发现一个新的问题。这适用于代码审查吗？

这是一个很重要的问题，因为自我检查的时耗最多只有引入他人审查时的一半。

Cisco的一个研究比较了300个进行过自我检查的代码审查结果和300个进行非自我检查的代码审查结果[1]。如图18-5所示，自我检查过的代码缺陷密度是非自我检查的一半，这表明人们在检查自身工作时发现了一半的缺陷。

只有简单的“自我检查”对于你自己来说是否充分，见仁见智。毕竟，其他审查者还是发现了很多问题。但有一点是明确的：自我检查有效地利用了时间，并且比没有自我检查好很多，因为它在没有任何外部审查时就能发现一半缺陷。

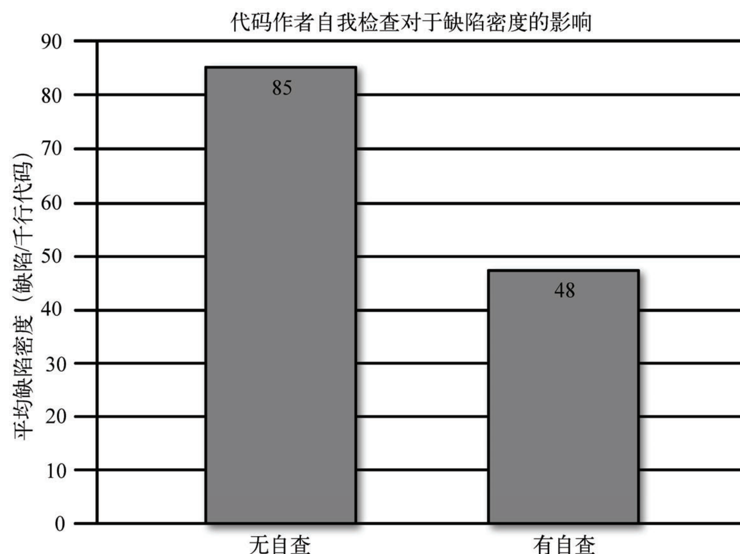


图18-5　当程序员审查自己的代码时，他们会发现其他审查者发现缺陷的一半

18.4　结论

对于所有的软件开发流程来说，问题不是“代码审查是否必要”，而是 “什么时候进行代码审查是有效的？最佳实践的方法是什么”。

这份研究报告给出了最佳实践的方法，不必再有条不紊地、无意识地浪费时间。

18.5　参考文献

[1] [Cohen 2006] Cohen, Jason. 2006. *Best Kept Secrets of Peer Code Review*. Beverly, MA: SmartBear Software.

[2] [Conradi 2003] Conradi, R., P. Mohagheghi, T. Arif, L. Hegde, G. Bunde, and A. Pedersen. 2003. Object-Oriented Reading Techniques for Inspection of UML Models—An Industrial Experiment. *European Conference on Object-Oriented Programming ECOOP'03*: 403-501.

[3] [Dunsmore 2000] Dunsmore, A., M. Roper, and M. Wood. 2000. Object-Oriented Inspection in the Face of Delocalisation. *Proceedings of the 22nd International Conference on Software Engineering* (*ICSE*) *2000*: 467-476.

[4] [Fagan 1976] Fagan, M. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15(3): 182-211.

[5] [Jazayeri 1997] Jazayeri, P., and H. Schauer. 1997. Validating the defect detection performance advantage of group designs for software reviews: report of a laboratory experiment using program code. *Proceedings of the 6th European Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Zurich, Switzerland, September 22-25, 1997): 294-309.

[6] [Kelly et al. 2003] Kelly, D., and T. Shepard. 2003. An experiment to investigate interacting versus nominal groups in software inspection. *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Centre for Advanced Studies Conference*: 122-134.

[7] [Votta 1993] Votta, L. 1993. Does every inspection need a meeting? *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, December 8-10*: 107-114.

第19章

公共办公室还是私人办公室

Jorge Aranda

本章涉及的问题似乎很简单，怎样来安排办公室的空间对软件团队来说最合适？我们一般认为项目的成绩好坏是由团队成员之间的距离和互动的难易程度决定的。小隔间办公室、家庭（电话）办公室、公共办公室以及其他的一些办公室的安排都在交流方式上有着各自的取舍，而在有些极端情况下（比如成员距离较远的分布式团队），要想进行交流互动几乎只能靠电子通信。当然，我们也知道，就算团队的成员在同一个城市的同一个办公室内，办公室的安排布置也会影响他们的效率。那么，到底办公室的安排布置有多重要？我们目前又掌握了哪些有助于更好地设计办公环境的研究结果？

19.1　私人办公室

有一种说法是：花钱来改善软件开发人员的工作环境，那些钱就是白白地打水漂。那些想要拥有独立办公室，可以关门、关掉电话的开发人员既自负又傲慢，这群人的欲望永无止境。如果你真的给每个人一间漂亮的办公室（先不说哪儿来的钱和空间），他们又会开始抱怨茶水间不够高档，要么就是自行车室内停放处的泊位太少，要么就会抱怨显示器屏幕上会反射窗户上的光。最后，他们不免会要求公司供应有机餐饮、本地出产的食物，并要求公司配备按摩师，再安上海洋球池，就像那些Google的员工们一样。即便开发人员们不会那么过分，为每人安排一间办公室的成本也很高，比小隔间高很多。为什么公司要把钱花在成效甚微，甚至可能根本没有效果的事情上呢——尤其是当大部分的竞争者都没有这样做的时候。

可能你已经知道了，但我仍然要说其实这样做有一个非常重要的理由。任何做过软件开发的人都知道这项工作有多复杂，对人的意识要求有多高。有时候开发人员面对的问题是如此的错综复杂而又高度抽象，他们必须考虑各种影响以及后果，所以他们常常需要全身心地投入到工作之中，直到其他所有东西都开始变得模糊，时间悄悄溜走，此时此刻他们已和技术合二为一。Csikszentmihalyi把这种深度专注的最佳状态称为*心流*（flow），即人的行为、技能和目标达到了和谐状态，并让自己充分沉浸于这种状态之中的体验。如果你听说过这个概念，那你可能也知道要达到这种“心流”状态需要时间和努力，而且这个状态非常脆弱，对于进行创造活动

的人来说极为重要。

如果开发人员不断受到各种干扰（比如旁边卡座的交谈声、电话铃声、窸窸窣窣吃东西的声音或者食物的味道等），他们就没办法工作。当然他们自己也会意识到这一点，所以不但事情做不好，最后还会因此变得很沮丧和失落，最后会导致更多的问题和错误。

不幸的是，在通常的公司环境中开发人员难以达到心流状态。有很多研究报告称开发人员经常受到各种干扰，其中我最喜欢的是由Andrew Ko、Rob DeLine和Gina Venolia做的一份研究。①他们在90分钟的时间内观察，到底是什么情况让开发人员们把注意力从手头的工作上移开，要么是因为他们的同事过来打断他们，要么就是因为他们缺少一些必要的信息所以被堵在那儿了，而此时又可能要去打断他们的同事[11]。

这些研究人员发现，对于这些开发人员来说，工作的进行是非常零散的。他们不断地被打断，而且他们也不断地打断别人，以便能找出需要的信息。他们会为Email、即时消息和参观办公室的人所分心。这些调查结果让人担心的地方还不止如此——调查是在微软进行的，要知道为每个开发人员提供私密而安静的空间正是让微软引以为傲的优点。可以想象，普通使用小隔间或者公共办公室的软件开发组织中，问题可能比这个严重得多。

在知道了这样的一些结果之后，我们开始着手设计一套办公室布局，希望能尽量满足需要独立工作的开发人员。因为我们推论：独立工作就意味着较少的干扰，也就意味着更好和更深的专注，那么就能达到更高的效率和更好的质量。这样的办公室有什么特点？首先，应该保证每个开发人员都有单独的办公室，空间小一些也行，而且要有可以关闭的门。当然，他们还可以把电话设置成静音，而且无需从上班到下班一直在线，当出现干扰的时候，他们有权自行选择如何处理。当需要团队交流的时候，开发人员可以使用共享的空间，比如会议室。此外，他们还应该有休闲的空间来进行社交活动或者稍作休息以便进入下一段的专注状态。这样，他们就能通过独立工作进入深度专注状态，随之而来的还有很多的好处，比如更高的效率、更好的产品质量以及更高的团队士气。

理论不错，但是有没有证据可以证明？碰巧的是，有。这个证据正是Tom DeMarco和Tim Lister的软件开发名著《人件》的核心论点之一。

在这本书中，DeMarco和Lister描述了他们举办多年的软件开发人员竞赛，题目是“在最短时间内实现一系列的基准打分以及测试代码并保证最小的故障率”。他们把这些竞赛称为“编程战争游戏”（Coding War Games）。参与者们必须记录下他们使用的时间，在他们宣布任务完成之后，产品必须经过验收测试。开发人员们“工作在他们自己的工作区域，在正常的工作时间进行，并且编程语言、工具、终端和计算机与他们在其他项目中使用的相同”。

DeMarco和Lister发现“竞赛选手们之间的差异巨大”，而且他们还发现了三条经验法则“似乎无论何时用于衡量绩效的区别都会适用”[8]：

- 最好选手和最差选手的成绩比率大概是10：1；

① 声明：我曾用整个夏天的时间在微软研究院做实习研究员，与Gina Venolia一起在Rob DeLine的编程中的人类互动（Human Interactions of Programming）小组中工作。

- 最好选手比中等选手（即成绩为中位数）的成绩大概好2.5倍；
- 中等选手以上的那一半的总成绩与另一半的总成绩的比率大概是2∶1。

更重要的是，DeMarco和Lister发现，成绩的好坏不是由编程语言、工作的经验年限、年薪或者曾经提交的故障数量来决定的。反而与工作的场所和成绩的好坏关系密切：高效的选手通常来自同一个公司，而这个公司的环境通常会比低效选手的公司环境要好。他们将这些结果制成表格，我们转载于此，即表19-1。

表19-1 代码战争游戏中好选手和差选手的工作环境对比

环境因素	排名在前25%的选手	排名在后25%的选手
你的专属工作区域有多大	78平方英尺	46平方英尺
这个区域是否足够安静	57%的选手答案为是	29%的选手答案为是
这个区域是否足够私密	62%的选手答案为是	19%的选手答案为是
你的电话能否调成静音	52%的选手答案为是	10%的选手答案为是
你能不能把电话转接给他人	76%的选手答案为是	19%的选手答案为是
你是不是经常毫无意义地被人打断	38%的选手答案为是	76%的选手答案为是

De Marco和Lister认为这些惊人的结果只能证明二者有相关性，但是无法证明有因果性。正如他们所说的："这里发表的数据并不一定可以证明更好的工作环境可以带来更高的效率。它也许只代表了表现更好的人更容易被提供更好工作环境的公司所吸引。这对你来说真的有意义吗？"

这些关于私人办公室以及最大限度免除打扰的工作环境的研究结果让人心动不已，而且办公空间布置上的额外成本也由增加的效率补偿了。但是故事还没结束。虽然看似圆满结局，但是却突然杀出个程咬金，让我们的研究进入了一个完全不同的方向。

19.2 公共办公室

我们前面的讨论有一个问题，那就是在DeMarco和Lister的研究中，每个人都"独自"完成任务。他们独自编写代码和进行测试，而且他们也是按个人来排名的。要想做好开发，唯一的关键就是"一个人"尽量集中精神，并找到最佳方法。

不过大多数的软件开发并不是如此。开发人员们并不是单兵作战，每个人都是团队的一份子。很多成员的工作都有交集。他们相互帮助，共同进退。所以，在项目进行的过程中，团队中的人们都需要不断地相互沟通和协调。

这种沟通和协调的需求可能会非常大。正如Brooks在《人月神话》[5]中指出的那样，对于沟通协调的需求太大了，以至于在项目后期加入新的成员可能反而会推延交付日期，因为此时加入新人会造成很多额外的协调开销。

对协调问题的思考使我们不得不重新审视"私人办公室"的解决方案。也许真正重要的是让开发人员们有一个可以一直高效地相互协调和交流的环境。比起专注来说，也许我们更应该强调协调。

有很多理论可以支持这个论点。首先，我们必须记住在Csikszentmihalyi的理论中并没有说心

流状态只能在孤立的情况下才会产生。事实上，和谐的团队精神是最容易进入心流状态的途径之一。强调协调还有其他的好处。比如，社会心理学家们发现，有亲密关系的人们会形成一种被称为“交互记忆系统”（transactive memory system）[15]的伙伴关系。你可能和你的伴侣或者家人经历过这种感受：有的人（很可能就是你）成为了“电脑专家”，有的人会专门负责记录大家的生日，有的人负责账单，等等。在交互记忆系统中的人们往往会潜移默化地确定每个人的专长。只要专长的分工是平衡的，那么这样的安排就会很高效。人们不必知道每件事情怎么来做，他们只需知道应该问谁就行了。在不考虑外界干扰的情况下，一直保持亲密关系的人组成的小组在记忆测试中的成绩会比陌生人组成的小组要好。这样的一个效应不但存在于个人关系之中，也存在于工作关系之中[10]。所以，如果团队的成员们相互坐得很近，而且可以不断地回答队友的问题，那么他们就更容易发展出一套交互记忆系统——无需刻意，只需自然而然。

这不仅仅意味着团队发展出了所谓的“团队记忆”。在我们开始思考如何进行高效的协调和沟通之后，其他的一些因素映入了我们的眼帘。一个因素是我们所使用的沟通渠道的丰富性。根据两位Olson的说法，虽然我们现在有很多技术可以帮助我们进行远程协作，但是距离仍然是个问题，没有什么比面对面的合作更高效。事实上，他们还进一步地预测，在接下来的数十年里，距离的问题仍然会很重要——即便我们实现了所有合理可行的远距离协作技术。当谈话双方处于同一个环境下的时候，双方交换的信息比他们有意识想要表达的信息要多得多，而这些信息的通畅对工作效率和质量都有很大的影响[12]。

Herbsleb和Grinter曾研究过一个项目，由于无法进行即时的交流，最后出现了问题。他们研究了一个成员处于不同地域的分布式开发项目在整合代码上的困难，并总结说整合代码是这个项目中最困难的部分，因为非正式交流渠道的中断造成了很多协调的问题[9]。他们的研究报告并没有说这个经历是超乎寻常的。

他们的研究适用于任何成员分散得很远的项目，但是我们也可以提出疑问，即如果人们在同一办公地点的不同办公室工作，非正式的交流渠道也会出问题吗？如果使用独立办公室的话，非正式的交流渠道真的会受那么大的影响吗？

确实会受影响，但是并不会像不同地理位置那么大。这里有一个问题，那就是如果人们离你的办公桌超过了简短步行的距离（大概30米），你就可能当他们在另一个城市，也就是说，你去找他们聊天或者问问题的机率就会直线下降[1]。两位Olson列出了在工作地点不同时，很多其他造成交流失灵的原因：他们没办法迅速获得反馈、没办法利用更多的交流方式、没办法获取微妙的信息、没办法了解很多隐性的暗示，等等。当人们常常不在同一个房间的时候，他们的交流的效率就要低一些。他们的凝聚力也会变差，而我和我的同事们曾推论凝聚力可能是软件开发组织成功的重要因素[2]。

敏捷思想的追随者们总是强调这几点。他们呼吁持续不断而且足够充分的互动，而大部分甚至所有的敏捷方法都强调了团队配置的重要性。尤其是Beck的极限编程方法，这种方法明确地推荐了各种相关实践，比如在整个项目进行过程中团队应该“坐在一起”（Sit Together），工作地点都在“富含信息的工作空间”（Informative Workspace）[4]。有些敏捷的支持者认为（出于直觉或者出于经验）：虽然相互配合会分散注意力，但是仍然比孤军作战要好。可是，有什么直接证据

可以支持这种说法吗？

我见过最有力的证据来自于Teasley等人做的一份研究。在这次研究中，他们调查了一家入选《财富》百强的汽车公司，这家公司愿意尝试“作战室”[①]的做法。这家公司先是做了一个试点项目，后来将这个做法推广到了公司中的大部分项目。作战室为每位项目成员都配备了电脑，大部分人都围着一个大办公桌坐，墙上挂着白板。旁边还有一些私密的小隔间，备有电话和电脑，如果成员需要安静或者隐私的时候，也可以选择这些小隔间。研究人员们追踪调查了数个效率的指标，以及团队成员们对于作战室的满意程度[13]。

试点团队的结果出奇地好：他们的效率超出了公司标准的两倍，虽然也有少部分成员不满，但是大部分还是很喜欢作战室。他们对于相互之间越来越密集的互动逐渐适应，并且认为这样的办公室比以前的旧式小隔间办公室要好。在这个做法被制度化了以后，后续团队的成绩比试点团队的“更好”，在保持团队满意度不变的情况之下，所有的效率指标都得到了长足的提升。这种激进的安排方式取得了成功，这一点和其他的科学研究中报告的一致。

你可能会想，那还有关注度和心流状态的问题呢。之前不是说公共办公室会造成很大的干扰吗？Chong和Siino研究了这个问题：他们对比了结对（而且紧密地安置在一起）的程序员和单个程序员的干扰问题。研究发现，在结对工作环境下的干扰有所不同。比起单独工作的人来说，结对编程的人们受到的干扰时间更短，受到的干扰和工作的相关性也更高。结对编程的程序员们所处的公共办公室确实要比单个程序员的小隔间办公室要嘈杂一些，但是结对程序员们都“适应了工作环境，只有在其他人高声喧哗时才会受到干扰。”Chong和Siino还发现对于独立工作的程序员来说，只要有人走到小隔间来，就会影响到小隔间内的人，无论这个人当时是在干什么。相比之下，公共办公室环境下的结对程序员们会告诉可能打断他们的人什么时间来最好，什么时间他们在做事不要来干扰。

就这些证据来看，这样的公共办公室确实是有好处的。团队坐在同一个办公室，全部人都围绕在一个或者两个大型办公桌前而不必分区，宽阔的墙上放着白板，还有少量私密办公室供那些确实需要隔离的人们使用。看来，这种布局可以最大限度的发挥他们的协调和认知能力。

19.3 工作模式

那么到底是怎么回事？结论有点自相矛盾：我们先是提到了独立办公室的好处，并称开发人员应该尽量集中注意力，后来我们又把论点转到了聚拢和公共空间的好处上，又称开发人员应该尽量多考虑协调能力。有哪个环节出问题了吗？

也许“工作模式”可以提供一些线索。根据Tesluk等人的研究来看，团队的工作流程可以被分为数个类别，这个分类基于团队成员所需要参与的互动类型。具体分类如下：

- 汇总式工作流程

  每个人的任务将会汇总到团队的级别。在这种模式中，团队成员之间无需交互或者交流。

---

① “作战室”这个词有种激进的意味在里面，所以我更喜欢使用“公共办公室”这个说法，不过二者的意思都是一样的。

- 顺序式工作流程

  任务从一个成员传递到下一个成员，但是不会反过来。

- 互惠式工作流程

  和顺序式流程类似，任务由一个成员传给另一个，但是这次是双向的。

- 密集工作流程

  任务将会有可能经手所有的组员，而全体组员也必须合作来完成任务。

当然，这样的分类仅仅是理想分类，团队一般不会正好属于某一类。虽然说项目的具体分类必须考虑其实际的流程和操作，但是一般来说软件项目倾向于后两类。很少有软件项目可以采用*汇总*或者*顺序式*的工作流程。（顺序式工作流程的例子是纯粹的瀑布模式，即不允许回到项目前一阶段的开发模式。）但是*互惠式*工作流程则比较靠近现实世界中的瀑布及螺旋开发模式，而密集工作流程则更像是敏捷开发方法。

这一点很重要，因为据我们所知，工作流程越是密集的团队，越是需要现场的协调。在一份详细的研究文献中，Beal等人总结道，对于那些*工作流程非常复杂*的团队来说，其凝聚力与效率之间有紧密的关联，但是对于那些工作流程简单直接的团队来说则不一定[3]。

想想你上一个软件项目。是不是每个团队成员都对项目要求以及自己的任务有着清楚的认识？他们是否知道向谁来询问他们需要的信息？他们是否知道自己负责的工作该如何和其他队友的工作对接？有没有一个信息库可供他们查询信息以及这个信息库是否有用？如果回答是肯定的话，工作流程就偏向于顺序式，而团队的协调需求就较少。如果回答是否定的话，那么工作流程就偏向于密集式，协调需求也就越多。

我们还可以使用另外一个方法来看这个问题。想象一个项目，这个项目没有任何明确的要求，对于这个软件能做什么只有一个大概的想法。它也没有明确的完成日期，团队只需承诺尽可能多地持续向客户交付新的功能。此外，你和你的队友相互之间不甚了解，你们以前没有长时间合作过，而你也不知道每个人的优点和缺点。现在，想象每个成员都把自己锁在一间独立的办公室里，这样就营造了一个拒绝任何干扰（无可避免的除外）的环境。在这样的条件之下你怎么来做事呢？

现在，想象一个相反的状况：团队很明确地知道需要做些什么，规范分档非常全面而且制作仔细，每个人都分工明确，而且由于有了仔细的规划，整合应该也不会有什么问题。只需你和你的队友们每天花几个小时独立专注于自己的任务，这个项目就可以完美地进行，但是你却坐在一个公共办公室里面，不停地被干扰和打断。在这样的条件之下你怎么来做事呢？

可惜的是，还没有研究可以提供一个明确的准则来指导我们如何根据团队的不同来进行选择。从现在已有的证据来看，我们最多只能说，只要和团队的工作方式搭配得当，那么不管是“私人办公室”还是“公共办公室”，两种空间安排都可以带来好的结果。如果你致力于使用密集式工作流程（或者你当前的环境要求你使用它），那么你应该尽量满足密集的协调工作，而如果你更偏向于顺序式或者互惠式的工作流程，那么你应该保证开发人员能够专注于自己的工作，并将他们之间的联系削减到少数必要的事情上。

19.4 最后的忠告

关于“公共办公室还是私人办公室”这个问题，答案是“都可以”。也就是说，虽然二者完全不同，但是却都大大优于现在很流行的隔间农场式排布。

小隔间的问题是它拥有前面两种布置方式的种种缺点，但是却不具备二者的任何优点。首先，小隔间并不像私人办公室那样把开发人员们隔离开来，所以他们常常会被周遭的对话所干扰。但是它又不是真正的共享空间，所以人们也无法连续地进行协调交流，而这一点公共办公室可以做到。据我所知，对于软件开发来说，没有任何研究表明小隔间要优于其他的办公空间安排，我们应该尽量避免使用这种安排方式。要么选择专注，要么就选择协调。如果两者都不选那就亏大了。

19.5 参考文献

[1] [Allen 1977] Allen, T.J.1977. *Managing the Flow of Technology*. Cambridge, MA: MIT Press.

[2] [Aranda et al. 2007] Aranda, Jorge, Steve Easterbrook, and Greg Wilson. 2007. Requirements in the Wild: How Small Companies Do It. *Proceedings of the 15th IEEE International Requirements Engineering Conference* (*RE'07*).

[3] [Beal 2003] Beal, Daniel J., Robin R. Cohen, Michael J. Burke, and Christy L. McLendon. 2003. Cohesion and Performance in Groups: A Meta-Analytic Clarification of Construct Relations. *Journal of Applied Psychology*: 88(6).

[4] [Beck 2004] Beck, Kent. 2004. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley.

[5] [Brooks 1975] Brooks, Fred. 1975. *The Mythical Man-Month*. Boston: Addison-Wesley.

[6] [Chong and Siino 2006] Chong, Jan, and Rosanne Siino. 2006. Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. *Proceedings of the ACM 2006 Conference on Computer Supported Cooperative Work.*

[7] [Csikszentmihalyi 1990] Csikszentmihalyi, Mihaly. 1990. *Flow: The Psychology of Optimal Experience*. New York: Harper.

[8] [DeMarco and Lister 1999] DeMarco, Tom, and Tim Lister. 1999. *Peopleware.* New York: Dorset House.

[9] [Herbsleb and Grinter 1999] Herbsleb, James D., and Rebecca E. Grinter. 1999. Splitting the Organization and Integrating the Code: Conway's Law Revisited. *Proceedings of the 21st International Conference on Software Engineering.*

[10] [Hollingshead 2000] Hollingshead, Andrea B.2000. Perceptions of Expertise and Transactive Memory in Work Relationships.*Group Processes and Intergroup Relations* 3: 257-267.

[11] [Ko et al. 2007] Ko, Andrew J., Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. *Proceedings of the International Conference on Software Engineering.*

[12] [Olson and Olson 2000] Olson, Gary M., and Judith S. Olson. 2000. Distance Matters. *Human- Computer Interaction* 15: 139-178.

[13] [Teasley et al. 2002] Teasley, Stephanie D., Lisa A. Covi, M.S. Krishnan, and Judith S. Olson. 2002. Rapid Software Development Through Team Collocation. *IEEE Transactions on Software Engineering* 28: 671-683.

[14] [Tesluk et al. 1997] Tesluk, P.E., J.E. Mathieu, S.J. Zaccaro, and M.A. Marks. 1997.Task and aggregation issues in the analysis and assessment of team performance. In *Team performance and measurement: Theory, methods, and applications*, ed.Brannick, M.T., E. Salas, and C. Prince. Mahwah, NJ: Lawrence Erlbaum Associates.

[15] [Wegner et al. 1991] Wegner, Daniel M., Paula Raymond, and Ralph Erber. 1991. Transactive Memory in Close Relationships. *Journal of Personality and Social Psychology* 61: 923-929.

第20章

识别及管理全球性软件开发中的依赖关系

Marcelo Cataldo

全球性软件开发（GSD）正在普及。由于可以降低成本、招募各地的人才、开拓新市场，大大小小的公司都开始走上了GSD的道路。但是，想要让分布在世界各地成百上千的项目人员协同合作，说起来容易做起来难。实际上，在GSD的环境中协调设计和开发工作是极具挑战性的。这方面例子很多：有很多产品被扼杀在了襁褓之中，还有些花费了双倍的时间才完成。值得庆幸的是，在过去的二十年中我们得到了很多关于如何更好地利用GSD的经验教训。

本章有两个目标。首先，我们将用各类实证研究证明：要想有效地协调GSD环境下的开发工作，就必须综合对软件项目的“技术”和“社会－组织”两个方面的认识。然后，我们将论述如何将这些研究结果转化为一套实用的方法，并运用到GSD中的各种角色和各个阶段。

我们将用“社会－技术一致性”（Socio-Technical Congruence，STC）这个概念将本章的各部分内容联系起来。那么，在开始之前我们先来介绍一下这个概念。STC指的是从技术环境中所产生的协调需求与项目的社会－组织结构可以提供的协调能力之间的关系。STC主张：当一个项目的协调需求和协调能力相匹配的时候，项目会取得更好的成绩，例如更好的产品质量或者更高的开发效率。

最后，我们还将介绍STC所推荐的一套标准来衡量协调需求、协调能力以及它们之间的关系。在本章中，我们将在实践环境中来讨论这些因素。

20.1 为什么协调工作对于 GSD 来说是挑战

大型的跨站点或者跨组织的项目已经存在了几十年了。在20世纪50年代那会儿，组织结构方面的研究人员们最先想要解决的问题是如何安排好各种工作。Galbraith、March、Simon和Thompson等研究人员声称，要想安排好工作，就必须尽量减少项目或者组织的任务之间的相互依赖性。他们的研究指出：在必须要做协调的时候，我们如果能预先知道任务包含哪些步骤和各个任务之间的相互依赖关系，就可以使用标准作业流程（standard operating procedure）一类的机

制来管理相互依赖关系并有效地协调工作。但是，当出现特殊情况或者任务的内容并不明确的时候，我们就需要更具弹性的协调措施了，比如项目成员之间的直接交流或者会议等。

后来，Malone和Crowston [1994] 又提出了其他的一些协调方法，例如用传统的软件概念来给工作依赖关系分类（如“资源共享关系”或者“生产者/消费者关系”等），并根据不同的类别使用不同的协调策略。

在技术领域中，模块化系统设计方面的研究也提出了非常类似的观点。比如，Parnas很早就指出：系统应该被分解为较小的单元（如模块），而这些单元之间的依赖关系应该尽量减少[29]。在这种情况下，依赖关系的表现形式就是每个单元或者模块所公开的接口。只要在项目开始的时候就明确地定义好这些接口并保持稳定，那么各个单元的开发就可以同时进行。如果需要做修改，那么我们可以启用协调机制（比如会议）来把有关各方召集在一起讨论一个大家都同意的接口修改方案。

前面讨论的是在今天的任何组织结构（包括软件项目）中都能找到的最老生常谈的一种协调机制。但是，软件开发中的协调工作还面临着独特而严峻的问题，这些问题主要来自于软件项目的技术和社会两方面之间的复杂关系。

我们都知道，在软件项目中，我们对于需求的认识和理解通常是随着项目的进行而不断改变、不断完善的。随着需求的变化，正在开发的系统的结构也随之变化和发展。这种不确定性以及变化和发展所带来的复杂性是一个很大的问题，使得了解不同的项目部件之间的依赖关系并有效地管理它们变得非常困难。

比如de Souza等人[16]和Grinter等人[20]在研究了多个软件项目之后发现，很多开发人员完全搞不清楚他们的修改会对系统的其他部分以及这些部分所对应的开发任务造成什么影响。这种协调不力的情况常常会使项目在集成和测试阶段出现各种问题，有时候还会出现很严重的问题。

软件项目中存在的协调问题在全球性项目中更是严重[22]，原因如下。

- 地理上的间隔使得项目成员们无法简单地下楼和其他成员进行交流、讨论问题或者了解软件变更的情况。
- 时区的差别也会妨碍信息的交换和有效的协调。
- 当无法进行同步的沟通（如电话会议）时，人们就不得不使用异步的交流方式（如Email），这更容易导致错误或者误解。
- 一个项目所涉及的部门或者办公地点越多，组织上的障碍（如不同的管理、激励机制和目标）就越容易造成项目成员沟通和协调的不畅。

总之，我们必须认真对待软件项目中的协调问题。此外要注意的是，GSD的固有特点还会加重这个问题。

20.2 依赖关系及其社会/技术二重性

在设计和实现复杂的软件系统时，我们需要考虑各个组成部分之间的各种依赖关系。而开发人员、管理者和其他利益相关者通常无法识别和处理所有这些依赖关系。这些问题通常会导致较

低的生产力，因为他们需要更多地返工，还需要更多的时间来进行集成和测试[11][13][21]。我们也常常看到其对软件质量的影响，比如未被发现的依赖关系会导致更多的缺陷[5][11]。

在软件开发中，所有人都会面临的严峻挑战就是识别依赖关系。这并不容易，因为这涉及两个相关的方面：一个是技术部分，一个是社会－组织部分。比如，有时候依赖关系的技术本质会让其难以被发现。一个典型的例子就是在两个组件之间的异步远程调用（ARPC）。ARPC会产生时机、锁定以及资源消耗方面的依赖关系，但是这种依赖关系可能只有在出了相关的问题之后，才会被开发人员留意到。

还有一些依赖关系是在社会－组织方面的，和组织内部的工作安排及执行方式有关。比如，组织上常常会基于开发资源的可用性来分配任务。这样就可能造成任务之间的依赖关系，比如在世界上两个不同位置，甚至是几乎没有共同的工作时间段的两个地区（即时差大于7－8个小时，比如美国和印度）的人员对信息共享的需求。在这种情况下，这些依赖关系的问题就无法在工作时间解决了。

为了说明识别软件项目中依赖关系的一些难点，我们可以参考一下Bass等人提出的这个在众多大型开发项目中有着典型意义的例子[2]。1号系统是一个软件平台，用于支持一系列实时嵌入式产品。由于平台的技术需求范围广泛（支持各种功能、内存需求、时间需求，等等），可以预见的是新的功能将会定期地被加入进来。

架构团队使用了在此前的项目中已经成功实践过的各种方法来设计1号系统。这个系统由各个组件组成，并使用了组件集成框架，最小化了耦合度，并且由于有了定义良好的接口为基础，使得团队之间可以相互独立地进行开发。开发团队必须严格按照接口规范来开发组件。

架构团队根据以往的经验进行设计，并使用了关注点分离（separation of concerns）的原则来应对性能和资源利用方面的需求。例如，随机访问内存（RAM）被分为了数个逻辑分区，分别分配给了系统的不同部分，再用给予优先级的调度程序来满足性能的需求。也就是说，整体的设计遵循了广泛认可的原则和习惯。

组织结构方面，这个项目涉及四个开发站点，两个在德国，两个在法国。各个站点中最有资历的代表们组成了一个核心架构团队，每个站点都将负责一个或多个子系统的设计和开发。项目由其中一个德国开发站点中的项目经理进行管理，他频繁地在各个站点之间穿梭。每个站点的工程师数量都差不多。

尽管如此，Bass等人还是阐述了这个项目遇到的很多严重的问题，我将集中论述其中的两个。对于识别技术性和任务性依赖关系的困难程度，这两个问题可以作为很好的例子。

第一个问题是，在完成了第一个版本的时候，系统出现了严重的性能问题，尤其是在系统启动的时候。架构团队根据以往的经验，认为调度程序的优先级规则足以让各个团队独立工作。但是，他们没有意识到这些规则需要进行大量的调整才能适用于现在这个项目。要解决性能问题就需要大量额外的协调工作。

第二个问题是，组织结构妨碍了为解决性能问题而进行的协调工作。架构团队将组件及子系统中的很多设计决策权交给了相关的站点，而各站点所做的决策所带来的巨大影响一直等到开发后期才被发现。在发现了依赖关系之后，再想有效地协调就变得困难起来，因为团队分布在各地，

而且相互之间对于所做的工作也不了解。

这个例子说明项目的技术和组织方面并不是相互独立的。恰恰相反，他们互相影响。在接下来的这一节中，我将论述：第一，各种类型的技术依赖关系；第二，如果不能识别这些关系并不能协调解决的话，会对生产力和质量造成什么影响；第三，识别这些依赖关系的传统方法。然后，我将用类似的方式概述几类传统的工作依赖关系。接下来，我将探讨依赖关系的社会-技术二重性及其在GSD的环境下对效率和质量的影响。

20.2.1 技术方面

技术类依赖关系是软件系统的各个组成部件之间的相互关系。在架构或者细节设计的时候，这些部件可以是组件、模块或者类。而在实现的过程中，我们关注的部件则是源代码文件。

软件工程师们通常把技术类依赖关系看做是*语法*（syntactic）上的关系。也就是说，这种关系的具体表现就是程序语言的元素，比如数据结构、函数或者方法调用等。另一种看法是把其看做软件部件之间的*逻辑*或者语义（semantic）关系。比如，实现同一个需求的两个组件在逻辑上是相关的。

“发布/订阅系统”代表着另一种逻辑（或语义）关系。这些部件之间的联系不是直接靠“一个模块调用另一个模块”这样直白的方式体现出来的。而我们的论述主要集中在语法依赖和逻辑依赖两种关系之间的区别，因为这种区别和它们的识别难度密切相关。

语法依赖关系这个概念源自编译器优化技术，其主要的目的是理解不同程序语句之间的控制以及数据流关系。使用的方法通常是从源代码或者某种代码的中等表示法（比如字节码或者抽象语法树）中提取关联信息。然后再分析各种语法单位，比如语句、函数或者方法，以找出数据相关（如某个数据结构由函数A修改然后又被函数B使用了）或者功能相关（如方法A调用了方法B）的依赖关系。

这种基于语法的依赖关系的分析方法用处很多，从用于检测缺陷的静态分析到帮助开发人员理解和调试代码的工具，都有使用。不幸的是，这种类型的关联信息有一些问题。在某些情况下，这些信息会不准确。例如，在诸如C或者C++一类的编程语言中，函数指针和条件性编译指令常常会给基于源代码的语法分析带来很多困难。在面向对象的语言中，由于有多态性（polymorphism）的存在，使我们几乎不可能在实际运行之前就弄明白两个类之间的关系。在另一些情况下，语法依赖关系信息又过于富余了，查出来的源代码文件之间的关系简直是千丝万缕，使分析的过程变得困难重重。

另外一个分析源代码文件中的技术类依赖关系的方法，是分析在同一个开发动作（如开发某个新功能或者修复某个缺陷）之内同期修改的源代码文件的集合。Gall等人指出，同期修改的文件（比如在同一次版本控制操作提交中修改的文件）之间有着某种依赖关系。他们称这种关系为*逻辑依赖关系*[19]。很显然，逻辑依赖关系和语法依赖关系并非完全不同的两种关系。事实上，逻辑依赖关系不但包括了语法上的关系（比如某次提交是因为修改了某个函数的参数，这个函数由文件A中实现并在文件B中调用），还可能包括更复杂的语义上的依赖关系，比如某一个文件中所

进行的运算会影响另一个文件所实现的行为。

这种技术性依赖关系的分析方法有一大优点，那就是比起调用关系图（call graph）或者其他图表，它可以使我们更好地估计语义上的依赖关系，因为它不靠语言结构来确定源代码文件之间的依赖关系。比如，对于远程过程调用（RPC）来说，基于语法的分析虽然可以提供足够的信息来把两个模块联系起来，但是，这样的信息可能深藏于从RPC调用者到RPC占位方法（stub）再到RPC服务器模块的漫长路径之中。而用另一种方法分析就简单多了：调用RPC的模块和实现RPC服务器的模块实际上是同时被修改的，这样就建立了一个逻辑依赖关系，即被影响的源代码文件之间的直接依赖关系。

这种分析方法可以使我们更好地分析发布者/订阅者系统以及基于事件的系统。在这些系统中，调用关系图无法将相互独立的模块联系起来，因为从语法上来看它们并没有任何可见的关联。比如触发某个事件的模块和处理该事件的模块，它们之间就没有可见的语法关联。此时，这种分析方法就可以派上用场，用来分析系统中重要但是又不可见的依赖关系。

还有一点很重要，那就是这种方法可以过滤掉开发人员不需要的语法关联。比如，语法分析会凸显各种基本的库之间的联系（如内存管理、打印功能），因为这些库包含了高度耦合的文件。然而这些库虽然有着很高的耦合度，但通常也稳定并且不容易出错。

需要注意的是，用逻辑分析法来分析技术性依赖关系也有它的问题。主要的问题是这些依赖关系是从历史数据（比如版本控制系统）中提取的。如果没有这样的资源，就无法有效地确定源代码文件之间的逻辑依赖关系。

虽然两种方法都有其局限性，但语法分析法和逻辑分析法都能提供有效的补充信息，并从多个层面帮助识别和管理软件开发项目之中的依赖关系。下面这一节我们将讨论两种类型的依赖关系对于软件开发项目的生产力及质量的影响。

1. 语法依赖关系及其对生产力和质量的影响

语法依赖关系是一种简单的工具，它可以帮助你了解传统的软件工程概念（比如耦合度和凝聚度），以及这些概念会如何影响软件开发的质量和效率。不过，在对耦合度和凝聚度如何影响软件开发的研究中，大部分是关于二者对质量的影响，而关于二者对于开发效率的影响的研究则相对较少。

Banker等人就发现，一个软件组件的语法依赖关系越多，与其相关的维护成本就越高[1]。维护成本的升高主要是因为没有意识到随着语法关系的增加，复杂度也会升高。

在20世纪70年代中期，当耦合度和凝聚度的概念产生之后，研究人员们花费了大量的心思来研究这些概念之间的关系及其对于软件质量的影响。这一系列的工作造就了大批的度量法，从最简单的量化语法依赖关系（如数据类型的引用次数或者某个源代码文件中函数调用次数）的方法，到复杂的尝试捕捉耦合的各种结构特征（如继承树的深度）的方法。部分度量法已在第8章中讨论过。

如果用一句话来非常简单地总结这些研究的结论，那就是，“软件实体（如文件、组件、模块）中的语法关联越多，其缺陷就越多。”不过，语法关联和软件质量的关系还不只是那么简单。最近的一份研究[14]指出，数据相关的依赖关系（如数据类型的使用）比起功能相关的依赖关系（如函数A调用函数B）更容易引起软件缺陷。对于这种区别，一个可能的解释是：比起功能之间的关联，

要了解数据之间的关联需要更多抽象思维，这样才能知道数据会如何随着程序的执行而改变。

语法依赖关系的结构也很重要。比如，Zimmermann和Nagappan使用了图理论的方法来计算从Windows二进制文件中提取出来的语法依赖关系网络，并发现，将这种关系网络和传统的测量法（如代码改动）相结合使用，可以有效地预测发布后可能产生的软件缺陷[37]。

2. 逻辑依赖关系及其对生产力和质量的影响

逻辑依赖关系这个概念比技术依赖关系要新得多，这也就很不幸地意味着：关于它对软件项目的质量和生产力的影响，我们的了解要少得多。早期的研究结果表明，逻辑和技术两类依赖关系对于质量的影响是相似的。但是，近期的研究找到了一些更有趣的结果。

我和我的同事[14]研究了截然不同的两个公司的两个大型系统，并发现：在预测故障这件事上，逻辑依赖关系比技术依赖关系有效得多。事实上，只要我们考虑逻辑依赖关系，语法依赖关系的影响就可以忽略不计了。这些研究结果对于开发人员来说有着重要的意义，因为它们表明，要想识别和理解依赖关系，就不能把重心放在明显而直白的语法依赖关系上，而得去关注那些不那么明显的关联。

其次，这些结果还表明，逻辑依赖关系也会影响质量。比如，在对那两个大型系统的研究中[14]，我们发现，对于那些和某个源代码文件（例如A文件）有逻辑依赖关系的文件来说，它们之间的逻辑依赖关系越紧密，文件A相关故障的发生率就越低。这样的结果表明，开发人员应该更好地认识逻辑依赖关系，比如一组文件之间的联系有多紧密，如何才能保证对系统的某个部分的修改不会使其他部分产生问题等。更有价值的是，我们发现逻辑依赖关系的结构比语法依赖关系的结构更重要。比如，我和同事Nambiar一起研究了一个大型跨国的开发组织，并证明了构件之间的逻辑依赖关系的密度是影响GSD项目质量产出更为重要的因素之一，但是对于语法依赖关系则没有此类证据[10]。

20.2.2　社会/组织结构方面

软件开发中的依赖关系还有其社会性的一面，即完成开发任务所必要的沟通、信息共享和协调需求。我们将这些依赖关系称之为*工作依赖关系*。对于那些参加过软件开发项目（尤其是大型开发项目）的人来说，较为明显的一点，就是项目成员要想有效地识别和管理所有这些工作依赖关系有很多障碍，比如不同的经验、组织结构、地理位置，以及时间安排上的压力等。

比如，当开发人员对于系统的经验有限的时候，他们常常难以理解他们在任务中所做的修改（比如确保满足某个方法的调用前及调用后条件）对于整个系统有什么潜在的影响。这些认知差距的结果，就是更低的效率（比如需要返工）或者更差的质量。

然而，“经验”并不单纯指对于技术或者系统的了解。熟悉与团队成员的合作会大大改善生产力和质量，因为对于同事的了解会促进信息的共享以及协调工作。比如，工程师之间常常会发展出一套相互协调的方法，慢慢地相互之间都能知道在什么时候用什么方式来向对方问问题或者共享信息。

关于管理和识别工作依赖关系，有一个重要却又时常被忽略的因素是项目及开发组织的结构。这样的结构横跨了多个元素，比如参与项目的一系列单位（即团队、部门等）、它们之间的

正式汇报路线（即汇报的上下级关系）、它们的行政和开发流程甚至它们的员工激励机制。所有这些元素共同发挥作用，使项目成员们形成了一套互动、协调和合作的方法。

将软件项目的开发分布到多个地点不利于信息的共享和集成。比如，我们都知道开发人员们常常会共享一些和他们当前任务相关的技术信息，而这种共享的场所通常是在走廊或者咖啡机旁，而共享的方式是简短而休闲的对话。当开发人员们处于不同的地点时，这样即兴的沟通就不可能了。相反，项目成员们很难一直注意其他人正在进行的任务、做的决定和遇到的困难。最后的结果就是协调失当、信息整合出问题，最终导致生产力和质量的下降[16] [20]。

然而，分散在多个地点除了无法进行即兴的沟通之外，还有其他的问题。大幅度的时区差异（超过6小时）会显著降低用同步交互（如电话或者视频会议）来实时解决问题的可能性。在这种情况下，项目成员常常会选择异步交流，比如电子邮件。可惜的是，Espinosa等人已经证明，由于异步交流的信息流向管理相当繁杂，使用这种方式往往会造成大量的误解和错误[17]。

此外，分散开发还有一个问题，那就是项目组成员不太可能相互认识。通常我们对在同一个办公地点并熟悉到一定程度的同事会共享更多的信息，而对于其他办公室或者其他地点的同事则少一些。在这种情况下，当同事们互不相识时，对于共享信息请求的回复可能就会不及时甚至被完全忽略掉了。而结果就是项目组的成员常常难以识别相关的工作依赖关系，特别是当发生意外的变化时。

最后，软件项目在时间安排上的压力会对项目成员识别和管理工作依赖关系的能力产生重大的影响。通常来说，时间安排压力越大，并行（而且可能是相互依赖）的开发任务就会越多。想象一下，某个功能B本来是要在功能A之后开发的，B依赖A。由于时间上的压力，两个功能必须同时进行开发。为了做到这一点，开发人员需要面对新的、更加复杂的协调需求。

比如，两个功能之间的接口可能会随着开发的进行而发展，这就需要不断地进行协调以避免集成问题和难以发现的缺陷。此外，开发人员可能还需要使用特别的“粘合代码”（glue code）来逐步测试正在开发的代码。换句话说，要想圆满完成这些任务，就需要一套适当的协调机制，这套机制必须能够让开发人员识别相关的依赖关系并对这些关系进行妥善的处理。

下面一节我们将讨论各种类型的工作依赖关系以及他们对于生产力和质量的影响。

不同的工作依赖关系及其对生产力和质量的影响

最常见的看待工作依赖关系的方式是模拟任务之间的关系。这些依赖关系主要集中在任务的时间优先级上（如任务A必须在任务B之前完成）。使用这种思维方式的项目采用了大量的工具来识别和管理这些依赖关系，从分析性和图表化的方法（如甘特图和PERT图）到各种支持这种工作流程的工具。

我们也许都会知道：如果我们不能认识到这些依赖关系并对其做有效的管理，开发时间就会被拖长。但是，其实我们还可以对这类的信息做更有趣的分析。我和我的同事[14]参考了在同一个发行版本中各个开发任务的时间依赖关系，并使用了一种社会网络分析法来分析工作流程依赖关系图，图中的节点表示开发组织成员，边则代表开发任务在开发人员之间的交接。然后我们将使用社会网络分析法来分析这类人与人之间的关系。我们发现，这些关系中有很多都需要双方付出大量的精力来维护。

这一点在工作流程的依赖关系中非常重要，因为它表明，处于项目核心位置的成员有更多额外的消耗而这些消耗使他们更容易不堪负荷，增加了沟通不畅的可能性，也使得软件面临质量降低的风险。实际上，我们的结论正好支持这一论点。结论显示，当（被）依赖度很高的人修改一个源代码文件时，这个文件更容易出问题。

时间（或者说流程）类依赖关系有一个变体，即不关注于任务的完成或者交接，而关注于信息的需求。举例来说，假设我们有两个相互依赖的任务：开发模块A和开发模块B，模块B会调用模块A的功能。在这种情况下，这两个任务的时间依赖关系就存在于“如何实现B对A的调用”这个信息之中了。有了这个信息，开发模块A的开发人员可以定义一个接口，并提供这个信息给模块B的开发人员。然后，这两个任务可以同时进行。

并行工程学（concurrent engineering）的研究主要关注相互依赖的并行任务的管理方法，这类研究指出：如果我们可以提前确定信息需求，就可以使用适当的协调机制来处理相互重叠的开发任务之间的信息类依赖关系。可惜的是，常见的软件开发项目的实际情况却并非如此简单。在项目的整个生命周期中，常会有几十个甚至数百个任务会产生重叠。在很多情况下，我们在进行任务之前并不能完全掌握这些任务的时间优先级以及任务之间的信息类依赖关系。事实上，由于需求会逐渐确定下来，而开发人员对需求的理解也逐渐加深，任务之间的依赖关系也可能会产生变化。

我最近做的一次研究[5]分析了某大型跨国软件公司的209个分布式开发项目，并发现开发任务之间的重叠程度越高（根据任务追踪系统的记录），这个软件的质量就越低。更重要的是，在软件生命周期的任何阶段，任务的重叠所产生的影响都是一致的。也就是说，任务的重叠不只是在项目里程碑快要到达（通常此时并行的开发工作猛增）的时候才对质量有影响。它在任何时候都会影响质量。这些结果是很有意义的，因为它们展示了追踪这些依赖关系的重要性，所以管理者和其他相关人士可以采取行动来解决这些不良影响。

第三种工作依赖关系来源于组织，作为一个特定的角色，能否建立有效沟通和协调途径来满足共享信息的需求。Nagappan等人使用传统的组织结构图所提供的信息构建了一系列的指标，并研究了在Windows组件和程序中，这些指标和故障的关系[28]。虽然他们的方法并没有专门包含工作依赖关系，但却代表了众多的组织现象的指标（代理），包括工作依赖关系的相关问题。他们的分析结果非常有趣。

比如，参与开发某个组件（或二进制文件）的部门越多，其质量就越低。此外，开发或修改组件的人员的组织层级差别也会对软件质量有负面影响。这些结果强调了跨组织（如跨团队、部门、地点等）的沟通和协调所面临的困难，以及这些困难可能对软件质量造成的不良影响。

软件系统的开发包括一系列的设计决策，有的在架构层面，有的在实现层面。这些设计决策有可能带来一些限制，而这些限制有可能在系统各部分之间建立新的依赖关系，也有可能变更甚至消除既有的关系。依赖关系的变更又可能带来新的协调需求，而这类需求又很难预先确定。

比如，有个任务需要修改某组件中的RPC的内存分配策略以改善性能。这样的修改可能会影响这个组件和其他组件之间的RPC交互的时机，从而使这些RPC的用户所做的某些条件判断失效。为了更好地了解如何捕捉这种动态的依赖关系，我和我的同事[11] [13]提出了一套社会-技术框

架，用于分析软件的逻辑依赖关系和开发工作的结构之间的联系。

“协调需求”是这个框架中的一个衡量标准，这个标准衡量的是，在已知各个开发任务所影响的系统部件以及它们之间的技术依赖关系的情况下，各个成员的工作在多大程度上依赖于其他成员的工作。关于这个标准的一个重要发现，是开发人员所需要面对的协调需求越高，其所修改的源代码文件的质量就越低。换句话说，随着开发的进行，从系统的逻辑依赖关系所衍生出来的协调需求越来越多，开发人员也就越来越容易引入bug。

20.2.3　社会–技术方面

软件开发，或者更普遍一点，所有的产品开发都包含技术和社会/组织这两个元素。到目前为止，我们讨论了在单独一个方面内的依赖关系。但是，技术和社会/组织这两个方面实际上是相互交织的，孤立地去考虑某一方就无法看到项目的全貌。

从社会–技术角度来看软件开发中的依赖关系其实非常简单。当任务之间产生协调需求和开发人员的实际协调工作相符合的时候，开发效率和软件质量就会提升。然而，社会–技术视角真正的重要作用在于它可以识别和追踪社会和技术类依赖关系之间的动态联系，对不同的技术依赖关系（如语法或逻辑依赖关系）进行精细分析，并调查软件开发中由技术依赖关系所产生的协调需求。

比如，我和我的同事[7] [11] [13]使用来自两个大型软件项目的多个软件储存库（如版本控制系统、故障追踪系统等）中的数据来展示：当工程师识别并处理了相关的协调需求的时候，软件的质量和开发效率都会有所提升。这项研究的另一个重要结论是：工作相关的协调需求，实际上常常是从逻辑依赖关系而不是语法依赖关系中来的。逻辑依赖关系通常会捕捉到语义上的、更为隐性的联系，而不是像语法关系那样直白的联系（工程师们可以直接靠看代码来识别这些联系）。

比如，逻辑依赖关系可以表示两个系统组件之间的发布者/订阅者关系或者时序关系。在这种情况下，语法结构常常无法提供足够的信息来帮助我们识别这种依赖关系。

在我们对能够决定工作依赖关系的那部分软件依赖关系进行调查的时候，还发现了一个相关的结论，那就是协调工作没有对应到相应的依赖关系是影响开发效率和软件质量的主要因素。

结合考虑技术和社会/组织两个方面也让我们可以更好地认识如何才能改善跨多组织的大型软件开发项目。举例来说，我在对一个大型公司的209个项目的研究中曾总结，跨项目的构架依赖关系常常带来较低的软件质量[5]。这样的结果表明，超越传统组织结构（如团队）的协调和意识以及在更大范围的开发组织内部提供支持是非常重要的。

此外，我还和Nambiar探讨了在多地分布式开发的软件项目中，技术上的耦合性是如何影响质量的问题[10]。我们的主要发现是，在项目外部的技术依赖（比如组件A和B相接，但是组件B在此项目之外被修改）越多，软件质量就越低。跨项目的逻辑依赖关系使得预计的缺陷数量上升了50%，这个影响的大小和某些传统的因素类似，如产出的代码数量或者能力成熟度模型①的流程成熟度。

① Capability Maturity Model，用于分析开发流程并进行改进的模型。——译者注

20.3 从研究到实践

在前一节中我们讨论了在过去差不多十年里所学到的关于在GSD环境下处理依赖关系的经验。我们现在将分析这些研究成果的实际意义。

20.3.1 充分使用软件储存库中的数据

在调查依赖关系如何导致协调问题，进而导致生产力和质量降低的研究中，有很大一部分从软件储存库中收集到的数据。这种储存库的例子有版本控制系统、缺陷和任务追踪系统、各类产品维基，当然还有源代码本身。毫无疑问，这类研究强调创建及维护软件储存库可能带来的巨大价值，因为它可以帮助我们评估影响软件开发项目的各种因素。那么，既然这类储存库在现今的软件开发组织中已经非常普遍了，我们何不使用库中的丰富数据来改善软件开发工作呢？抱着这个想法，本节将讨论几个使用案例。

软件开发组织中，最简单而又最有用的做法，可能就是将版本控制系统和缺陷/任务追踪系统中的数据连接起来了。如果这样做，我们就将得到开发任务和缺陷之间的明显联系，并能从版本控制系统中找到相应的源代码（或者其他部件）修改。在这本书中，第9章和第25章都研究了这个问题。

显然，这种储存库之间的联系可以让我们识别一些软件流程中可追踪的基本联系。最重要的是，它使我们可以应用一系列的分析法和工具来大幅改善开发组织有效协调的能力，尤其是在GDS环境下的协调能力。比如说，我和我的同事[11] [13] [7]设计了一个框架，它使用软件储存库中的信息来计算出各种指标（如技术依赖关系、改动量、工作量）以及找出协调的模式（如使用缺陷报告中的说明）。这份研究不但表明我们可以评估协调上的失误对于完成任务的时间以及软件故障的影响，还表明我们可以只基于这些可以从绝大多数软件开发项目中找到的丰富数据来进行可靠（而且可以轻易自动化）的分析。Wagstrom在开源软件项目中重复了同类的分析，进一步地证实了这类分析的可行性和有效性[35]。

软件储存库已成为了支持软件开发工作（尤其是GDS环境下的开发工作）的众多协作工具中的核心要素。这些协作工具借助传统的基础类工具（如ClearQuest、ClearCase、Subversion、Bugzilla和oMantis等）来提升软件开发项目的沟通和协调能力。而协作工具也有很多种类，包括：

- 工作区感知技术（参见参考文献[4]）；
- 专长推荐器（参见参考文献[25]）；
- 工件推荐系统（参见参考文献[15]）；
- 中断管理系统（参见参考文献[18]）；
- 项目仪表盘及可视化状态（参见参考文献[31]）。

最近的一些工具，如Tesserac[34]采用了社会–技术的视角，把技术和工作类的依赖关系用可视化分析的方法结合起来，以方便识别和管理。虽然这些工具中的很多都是在研究项目中使用的，但是它们中不少功能和理念都已经商品化了，比如IBM的Rational Team Concert和微软的Visual Studio。此外，还有一些开源的实现，特别是作为Eclipse平台的插件。

软件储存库还可以用来分析系统模块化的方式是否有问题，以便重组或者重构代码。一旦开发人员、软件架构师或者其他相关人员意识到技术依赖关系存在着某种规律，他们就可以使用专门的技术来减少这些依赖关系（尤其是逻辑关系）。例如，重构系统可以减少逻辑依赖关系，对软件的质量会有积极影响。

其他的一些代码重组的技术，比如由Mockus和Weiss[26]提出的分块法（chunking），就可以使用各种类型的技术依赖关系，使系统的结构更适合于GSD环境下的软件开发组织。分块法将源代码文件分成多个组，每个组包含多个（在逻辑上和语法上）联系非常紧密的源代码文件，而每个组和系统的其他部分的逻辑关系非常少。相同的方法可以用在工作依赖关系上。

20.3.2　团队领导和管理者在依赖关系管理中的角色

关于依赖关系的社会-组织方面及社会-技术方面的研究指出，团队领导和管理者可以在GSD项目的两个方面发挥重要作用，这两个方面是组织障碍和工作分配。组织障碍（如部门、场所以及项目等）常常会造成很多困难，使得相关人员难以有效地识别和管理相关的依赖关系。

对待组织障碍的传统方式是建立一套流程，这套流程提供各种基础措施来管理相互依赖的团队或者办公场所之间的依赖关系。但是，由于组织边界（如工作地点）的数量越来越多，需求或者功能的复杂度及不确定性也越来越高，流程常常不能满足这些需要，反而开始阻碍开发的进展。事实上，我和同事Nambiar已经证明了，软件开发越分散，开发流程对于质量的改善就越少[9]。这种改善的减少常常是因为流程是为本地开发（如单个场所或者单个部门）而专门设计和优化的，这就导致了各个开发场所流程之间的不一致。

团队领导和管理者可以帮助大家解决这些组织障碍。首先，他们必须意识到团队领导常常只考虑自己职责范围内的事情。其次，他们必须找出相互依赖的工作地点（或者其他组织单位）都应该了解的重要人员以及重要的信息流动，以促进不同场所的流程之间的统一。最后，对于促进各工作地点之间的信息交换，他们必须保持主动而开放的姿态。

Suzanne Weisband对于管理者在分布式工作环境中的角色已经进行了数十年的研究[36]。她的主要发现是，当管理者主动地收集各个地点中的项目信息，并将这些信息共享给其他地点的时候，项目的效率有显著的提升。

工作分配是依赖关系管理的另一个重点，它包括两个重要的方面：工作将在哪里完成及工作将于何时完成。

在将任务分配到不同的地点时，管理者们的主要考虑的通常是能力和成本。但是不能忘记社会-组织因素的重要作用，忽略这个因素对于生产力和质量造成的负面影响有可能会抵消本来分布式开发所能节约的成本。比如，在最近做的一些研究[5][8]中我发现，那些开发人员的分布极不均衡的项目（如地点A人很少，地点B人巨多）常常比分布均匀的项目多产生40%～50%的缺陷。

此外，项目中同时进行的开发任务的数量越多，其软件产品的质量就越差。实际上，工作量在整个开发周期中平均分布的软件项目与那些偶尔会有高度并行任务的项目相比，前者的缺陷数量只占后者的一半。与此形成对比的是，在对类似的项目进行的比较中我们发现，技术和领域相关经验只造成了软件质量中20%的区别。

综合这些结论我们可以看出，管理者在分配任务的时候还需要考虑两个重要因素：人员的均衡分布和执行任务的时机。

总而言之，团队领导和管理者的帮助可以让依赖关系（尤其是分布式项目的依赖关系）的识别和管理大大改观。

20.3.3 开发人员、工作项目和分布式开发

软件开发人员总是穿梭在技术–工作依赖关系的复杂网络中。比如，他们花费大量的时间来了解代码的不同部分之间如何互动、有什么关联，以及对于代码某个部分的改变会如何影响其他部分。尽管他们花费了不少精力，但常常还是会漏掉一些依赖关系。

本章前面已经讨论了各种帮助开发人员更好地识别和管理依赖关系的方法。第一个方法是利用逻辑依赖关系。首先，它可以提供语法依赖关系之外的重要信息。而且，开发人员还可以靠它来更好地了解不同源代码文件和模块之间的技术关联，以及某些修改可能造成的影响。当然，使用相关工具用来收集逻辑依赖关系信息并按照适当的方式来向开发人员展示也很重要。不过，传统的基础工具也可以派上用场，比如我们可以从版本控制系统中找出哪些文件或者部件是同时修改的。

第二个方法是保持对其他组件或者模块中和自己的任务有重叠的部分的了解。这些组件或者模块相关的设计/实现决策或者修改可能会对开发人员的责任范围产生重大的影响。开发人员往往把注意力（无可厚非地）集中在他们直接责任范围内的开发任务上，例如一个组件、部件、模块或者一组源代码文件。要想对其他开发活动保持一定程度的了解，有一个常用的方法是在缺陷跟踪系统中订阅某个团队（或者组件）的任务更新。不过这种方法很快就不够用了，尤其是在大型项目中。开发人员常常会被潮水一般的更新通知邮件淹没，慢慢地也就不再注意它们了。将依赖关系分门别类将有助于开发人员过滤复杂的依赖关系网络，并对相关的开发任务保持足够的注意力。

最后，在相互依赖的团队（尤其是GSD环境中的团队）之间建立社交往来，也是一个有效地收集和分享相关依赖关系的重要方法。在几十年中所进行的关于交流、分布式团队和社会网络的研究中有个一致的结论，那就是相互之间有着融洽关系的人们常常会更多更及时地分享工作相关的信息。这些研究所发现的建立社交关系的方法包括：在非工作环境中面对面的互动、使用社会网络工具、展示能力及合作态度等。

20.4 未来的方向

这一节主要论述可能对识别和管理软件项目中的依赖关系产生深远影响的三个研究方向。

20.4.1 适合GSD的软件架构

依赖关系的社会–技术视角强调了尽早识别相关依赖关系的好处，在识别了依赖关系之后，项目组即可采取适当的协调措施。软件架构已经成为开发流程不可或缺的一部分，它提供了各种

技术和组织类的信息来帮助识别依赖关系。但是，软件架构代表的是高度抽象的结构，这可能会妨碍正确识别相关的技术依赖，以及随之而来的重要协调需求。

越来越多的人使用标准化设计及建模语言（如UML）来解决这类问题，这些语言的解决方案是提取信息并用图论分析法（类似于软件储存库挖掘的方法）进行分析。然后我们就可以看到软件架构的“协调视图”，这个视图包含了软件的技术依赖关系以及进行开发工作的各组织单位之间的关联。

一个可能更有前途的未来方向是加深对于软件架构师的设计思路（尤其是和技术依赖关系、工作依赖关系有关联的设计思路）的理解。软件架构师很少在理想状态下做设计。事实上，架构师们总是面临着各类技术和非技术的限制。所以，他们的设计流程总是充满了各种权衡取舍。

举例来说，架构师们在做设计的时候需要考虑的约束包括：

- 老代码在重组系统时可能带来的影响；
- 老组织结构在重新安排工作时可能带来的影响；
- 与用户需求无关的修改所需的资源成本（如客户可能不愿意花钱来重构软件平台来改善其适应性）；
- 处理项目之间和客户之间的区别。

软件架构师们常常会发展出自己的灵感来源和评估方法。迄今为止，只有一套有限的设计准则存在，这套设计准则是从西门子 [32]、飞利浦[23]和摩托罗拉[3]的研究中派生出来的。但是，研究人员以及准则的使用者们的兴趣却与日俱增，希望能研究一套系统的方法来评估软件的架构及修改，这套方法将考虑技术和组织的约束条件，尤其是大型分布式开发项目中的各种限制。

20.4.2　协作软件工程工具

如前所述，研究人员们花费了大量的精力来关注软件开发的沟通工具、关注工具以及协调工具。最近的一些实证研究已经证明了这些工具对于小型团队的价值[33]。尽管这些结论都是正面的，但对于意识工具的实证评估只限于小型团队。

当前的合作工具对于大型团队或者大型项目也没有提供足够的支持。在这些环境中，技术和工作依赖关系的复杂网络所形成的关注机制产生海量的信息，使得协调的好处减少。规模和信息过量的问题凸显出了当前协作环境的重大限制。从传统意义上讲，关注工具的主要目的是为了记录分布式的团队或者项目中的协调信息。但是，这并不意味着就应该展示所有的协调相关的信息，尤其是那些无法采取对应行动的信息。

可以采取行动的信息包括两个重要方面：*相关性和及时性*。有证据表明，开发人员常常会随性地“顺藤摸瓜”，并基于代码中的依赖关系做自我协调，以尽量减少自己的修改对他人的影响（反之亦然）。比如说，我们知道有些开发人员会在提交修改之前发邮件给团队中的某些成员，告诉他们即将发生的修改以及可能的冲突。

但是，我们已经知道每次提交都进行自动化邮件通知的话很容易造成信息过量，然后人们就开始渐渐忽略它们了。如果关注信息可以只发送给会受到当次修改影响的人，其效果就会大大增强。此外，信息的需求常常是动态的，所以除了识别正确的人以外，也应该在正确的时间进行传

送。信息的需求是随着开发人员的任务环境而改变的。

在以后，我们将有希望看到新一代的工具，这些工具将在更加社会化的环境下结合新的分析和可视化技术，并模拟当前的计算机化社交环境（social computing environment），以处理规模和信息过量的问题。然后这些新工具就可以将社会网络的方法应用到研究人员们推荐的合作工具以及各种商业产品（比如IBM的Rational Team Concert）中，用于识别各种组织边界并解决这些协调的障碍。

20.4.3 标准化和灵活度的平衡

对软件流程的研究和对社会–技术一致性的研究有很多共通性，因为软件流程的一大重点就是分析和管理组织内的不同成员、工具和产品之间的关系。社区中正在进行的研究主要关注于加深对跨企业的合作和协调的了解，并提倡用更偏向于从社会-技术的视角来看问题。

现在，有各种框架（如社会－技术一致性[11] [13]）可以帮助我们评估有多个角色跨多个单位且有着不同等级的不确定性的软件开发组织中的开发流程（如变更管理流程）的协调效能。我们将可以用这种新的方法配合传统的流程模型分析法来设计、模拟和评估软件流程。

我们还有一些尚待解决的研究课题，包括使用社会－技术指标来改善软件设计的方法并使用新的流程和工具来进行开发，以及如何才能将流程、互动和协作工具相互结合起来改善协调工作。

20.5 参考文献

[1] [Banker et al. 1998] Banker, R., et al. Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study. *Management Science* 40(4): 433-450.

[2] [Bass et al. 2007] Bass, M., L. Bass, J.D. Herbsleb, and M. Cataldo. 2007. Architectural Misalignment: An Experience Report. *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*: 17.

[3] [Battin et al. 2001] Battin, R.D., et al. Leveraging Resources in Global Software Development. *IEEE Software* 18: 70-77.

[4] [Biehl et al. 2007] Biehl, J., et al. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. *Proceedings of the SIGCHI conference on human factors in computing systems*: 1313-1322.

[5] [Cataldo 2010] Cataldo, M. 2010. Sources of Errors in Distributed Development Projects: Implications for Collaborative Tools. *Proceedings of the 2010 ACM conference on computer supported cooperative work*: 281-290.

[6] [Cataldo and Herbsleb 2008] Cataldo, M., and J. Herbsleb. 2008. Communication Networks in Geographically Distributed Software Development. *Proceedings of the 2008 ACM conference on computer supported cooperative work*: 579-588.

[7] [Cataldo and Herbsleb 2010] Cataldo, M., and J.D. Herbsleb. 2010. Coordination Failures: Their Impact on Development Productivity and Software Failures. Technical Report CMUISR-2010-100, School of Computer Science, Carnegie Mellon University. *http://reports-archive .adm.cs.cmu.edu/anon/isr2010/CMU-ISR-10-104.pdf*.

[8] [Cataldo and Nambiar 2009a] Cataldo, M., and S. Nambiar. 2009. Quality in Global Software Development Projects: A Closer Look at the Role of Distribution. *Proceedings of the 2009 Fourth IEEE International Conference on Global Software Engineering*: 163-172.

[9] [Cataldo and Nambiar 2009b] Cataldo, M., and S. Nambiar. 2009. On the Relationship Between Process Maturity and Geographic Distribution: An Empirical Analysis of their Impact on Software Quality. *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*: 101-110.

[10] [Cataldo and Nambiar 2010] Cataldo, M., and S. Nambiar. 2010. The Impact of Geographic Distribution and the Nature of Technical Coupling on the Quality of Global Software Development Projects. Forthcoming in *Journal of Software Maintenance and Evolution: Research and Practice*.

[11] [Cataldo et al. 2006] Cataldo, M., P. Wagstrom, J. Herbsleb, and K. Carley. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*: 353-362.

[12] [Cataldo et al. 2007] Cataldo, M., et al. On Coordination Mechanisms in Global Software Development. *Proceedings of the International Conference on Global Software Engineering*: 71-80.

[13] [Cataldo et al. 2008] Cataldo, M., et al. 2008. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. *Proceedings of the Second ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*: 2-11.

[14] [Cataldo et al. 2009] Cataldo, M., et al. Software Dependencies, Work Dependencies and their Impact on Failures. *IEEE Transactions on Software Engineering* 35(6): 864-878.

[15] [Cubranic et al. 2005] Cubranic, D., G.C. Murphy, J. Singer, and K.S. Booth. 2005. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering* 31(6): 446-465.

[16] [de Souza et al. 2004] de Souza, C.R.B., et al. 2004. How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development. *Proceedings of the 12th International Symposium on Foundations of Software Engineering*: 221-230.

[17] [Espinosa et al. 2007] Espinosa, J.A., et al. 2007. Do Gradations of Time Zone Separation Make a Difference in Performance? A First Laboratory Study. *Proceedings of the Int'l Conference on Global Software Engineering*: 12-22.

[18] [Fogarty et al. 2005] Fogarty, J., A.J. Ko, H.H. Aung, E. Golden, K.P. Tang, and S.E. Hudson. 2005. Examining Task Engagement in Sensor-based Statistical Models of Human Interruptibility. *Proceedings of the SIGCHI Conference on Human factors in computing systems*: 331-340.

[19] [Gall et al. 1998] Gall, H., K. Hajek, and M. Jazayeri. 1998. Detection of Logical Coupling Based on Product Release History. *Proceedings of the International Conference on Software Maintenance*: 190.

[20] [Grinter et al. 1999] Grinter, R.E., et al. 1999. The Geography of Coordination Dealing with Distance in R&D Work. *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*: 306-215.

[21] [Herbsleb and Mockus 2003] Herbsleb, J.D., and A. Mockus. 2003. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Transactions on Software Engineering* 29(6): 481-494.

[22] [Herbsleb et al. 2006] Herbsleb, J.D., A. Mockus, and J.A Roberts. 2006. Collaboration in Software Engineering Projects: A Theory of Coordination. *Proceedings of the International Conference on Information Systems*.

[23] [Kommeren and Parviainen 2007] Kommeren, R., and P. Parviainen. 2007. Philips Experience in Global Distributed Software Development. *Empirical Software Engineering* 12(6): 647-660.

[24] [Malone and Crowston 1994] Malone, T.W., and K. Crowston. 1994. The interdisciplinary study of coordination. *Comp. Surveys* 26(1): 87-119.

[25] [Minto and Murphy 2007] Minto, S., and G.C. Murphy. 2007. Recommending Emergent Teams. *Proceedings of the Fourth International Workshop on Mining Software Repositories*: 5.

[26] [Mockus and Weiss 2001] Mockus, A., and D. Weiss. 2001. Globalization by chunking: a quantitative approach. *IEEE Software* 18: 30-37.

[27] [Nagappan and Ball 2007] Nagappan, N., and T. Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*: 363-373.

[28] [Nagappan et al. 2008] Nagappan, N., B. Murphy, and V.R. Basili. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. *Proceedings of the International Conference on Software Engineering*: 521-530.

[29] [Parnas 1972] Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12): 1053-1058.

[30] [Pieper et al. 2009] Pieper, J.H., et al. 2009. Team Analytics: Understanding Team in the Global Workplace. *Proceedings of the 27th International Conference on Human Factors in Computing Systems*: 83-86.

[31] [Ripley et al. 2007] Ripley, R., et al. 2007. A Visualization for Software Project Awareness and Evolution. *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*: 137-144.

[32] [Sangwan et al. 2006] Sangwan, R., et al. 2006. *Global Software Development Handbook*. Pennsauken, NJ: Auerbach Publications.

[33] [Sarma et al. 2008] Sarma, A., D. Redmiles, and A. van der Hoek. 2008. Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management. *Proceedings of the 16th International Symposium on Foundations of Software Engineering*: 113-123.

[34] [Sarma et al. 2009] Sarma, A., L. Maccherone, P. Wagstrom, and J.D. Herbsleb. 2009. Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development. *Proceedings of the 31st International Conference on Software Engineering*: 23-33.

[35] [Wagstrom 2009] Wagstrom, P.A. 2009. *Vertical Interaction in Open Software Engineering Communities*. Unpublished PhD dissertation, School of Computer Science, Carnegie Mellon University.

[36] [Weisband 2008] Weisband, S., ed. 2008. *Leadership at a Distance: Research in Technologically-Supported Work*. New York: Lawrence Erlbaum Associates.

[37] [Zimmermann and Nagappan 2008] Zimmermann, T., and N. Nagappan. 2008. Predicting Defects Using Network Analysis on Dependency Graphs. *Proceedings of the 30th International Conference on Software Engineering*: 531-540.

第21章

模块化的效果如何

Neil Thomas
Gail Murphy

在过去的30多年时间里，人们一直都把“模块化”视作大型软件开发的必备条件。模块化是指将系统分隔成多个小的软件单位（模块），便于个人或者团队进行开发[2] [9]，以此来控制开发的复杂度。模块隐藏了各种内部设计和实现，而只留出一个和其他模块联系的接口，这样负责各个模块的开发人员或者团队就可以相互独立地工作。模块将软件系统的各个部分强制区分开来，为它们提供相互沟通的渠道[2]，最大限度地减少软件系统中单个部分的修改对其他部分的影响[9]，并能在做出修改后减少编译的时间（各模块可以分别进行编译）[7]。

大部分程序语言都可以直接定义模块，以帮助开发人员更好地发挥模块化的这些好处。很多面向对象的程序语言，如Java [3] 等，都使用了抽象数据类型来支持对模块的细致定义。此外，语言的配套技术偶尔也能提供一定程度的模块支持。比如，Java有一个配套的技术叫做OSGi，这个技术可以将类分成一个一个的束（bundle），然后束与束之间通过定义良好的界面进行通信[8]。开发人员还常使用一些非正式的方式来表示模块，比如使用层级文件结构来区分各个模块。

在理想情况下，每个模块都应该包含一个关注点。举个例子来说，如果我们有一个控制火车的软件系统，那么开发人员就可以把与刹车互动的部分定义成一个模块，这样就能把刹车部分的相关逻辑都统一起来。不过在现实情况下，我们无法总是这么清楚地将各个关注点区分开来。再回到火车的例子，开发人员可能还会定义另一个模块来和轨道系统中的列车信号系统进行交互。有时候，刹车模块和信号模块之间可能会有非常紧密的交互，紧密到需要在刹车模块中实现信号模块的功能。例如，在接收到紧急信号后就必须立刻启用刹车。这样紧密的交互导致了更为复杂的模块化需求，也推动了面向侧面的软件开发思想（aspect-oriented software development） [5] [10] 以及相关工具（如AspectJ）的发展[5]。这些新的思想和技术提出了横切关注点（crosscuting concern）的概念，也让开发人员可以更好地对系统进行模块化。

模块化就此成为软件开发中的一种正式方法。不过，它的效果真的像宣传的那么好吗？在这一章中，我们将分析三个开源软件的档案，并回答以下问题。

- 是不是大部分的代码修改（如修复某个bug或者改善某个功能性）都被约束在单个模块内？
- 当开发人员对代码进行修改的时候，是否必须参考其他模块的代码？

- 我们是否可以根据实际的修改情况以及对其他模块的参考情况，为这套系统推导出一套新的模块化方案？

在开始调查之前，先介绍一下即将分析的几个软件系统。要想让我们的回答经受住不同软件系统的考验，就必须要知道：第一，什么样的修改才具备横向可比性；第二，对于这些系统都适用的模块定义。所以，我们将先定义“修改”和“模块”两个概念，再用这两个定义作为分析的基础，并详细介绍分析过程，最后总结分析结果。

21.1　所分析的软件系统

软件开发项目之间在很多方面上都可能有所不同，比如项目所在的领域、开发人员的专业程度、项目的规模以及程序语言的选择，等等。我们希望尽量选择差别大一些（即在多个方面都有所不同）而又具备档案信息的软件开发项目来进行分析。很多项目都可以达到这个标准。我们最终确定了下面三个项目作为研究对象：

- Evolution（http://projects.gnome.org/evolution），GNOME桌面环境内置，集电子邮件、地址簿和日历于一体的应用程序；
- Mozilla Firefox（http://www.mozilla.com/firefox），受欢迎的跨平台浏览器；
- Mylyn（http://www.eclipse.org/mylyn），Eclipse编辑器的面向任务的界面，内置于标准的Eclipse发行版中。

表21-1展示了这几个项目在开发时间、主要语言、模块的数量（见21.3节）、代码行数以及修改的数量（见21.2节）上的区别。我们只计算了我们在本次研究中分析过的修改数量。

表21-1　各系统概览

项　　目	第一版发布时间	主要语言	模块数量	大概的源代码行数（SLOC）*	修改数量
Evolution	2001年12月	C	43	300 000	1939
Firefox	2004年11月	C++	45	4 000 000	11 710
Mylyn	2006年11月	Java	18	675 000	3055

*源代码行数（SLOC）是由cloc工具（http://cloc.sourceforge.net）来测量的。

这个概览表可以让我们对这些系统有一个大致的了解，但这个表并不能让人看到全貌。特别是这些统计将系统档案当作静止的数据来分析，使我们无法了解开发人员是如何随着时间的推移来对系统进行修改的。为了更好地理解各系统在开发上的区别，我们还分析了它们被修改的速率。

我们分析了各个系统每天修改的代码行数，以此作为修改的速率。图21-1按时间的推移展示了项目的修改情况。图中的点代表从项目的代码库中得到的当天的代码修改数据，点的高度代表了当天修改并提交到代码库中的代码行数。我们可以看到，Evolution的特点是前期有少量修改，文后慢慢地进入了长时间的高度修改期。Firefox的修改没有那么频繁，而且在某一时间点之后几乎没有了修改，这是因为开发人员们完成了3.5版的工作，并转移到下一版的开发上去了。Mylyn的修改显示出了某种间歇性的激增。

我们还可以通过累计数据得到类似的结论。图21-2展示的就是Evolution的累计修改数据，我们可以很明显地看出早期的缓慢增长和后期的持续增长。其他系统的累计数据也是类似。

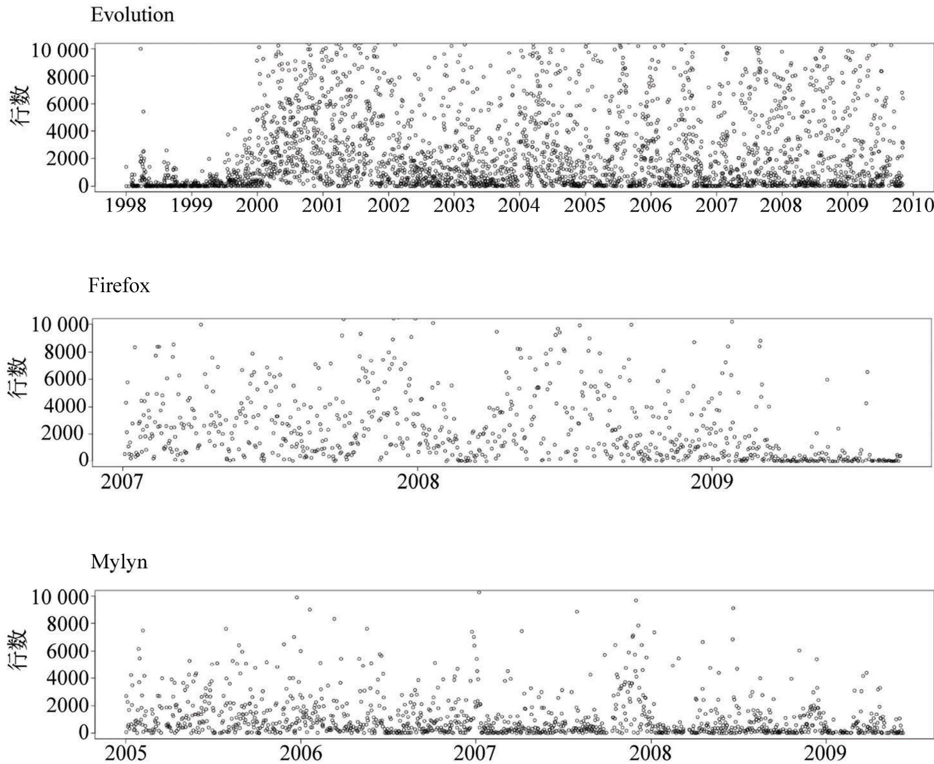


图21-1　三个系统每日修改代码行数的历史数据

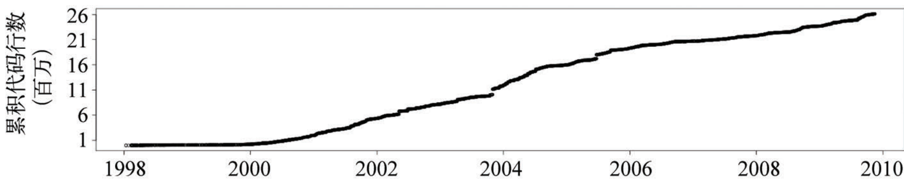


图21-2　Evolution的累计修改代码行数

21.2　如何定义“修改”

在我们开始分析模块和修改之间的关系之前，我们需要对“修改”做一个统一的定义。

我们所研究的这几个项目都使用Bugzilla（http://www.bugzilla.org）作为主要的bug追踪系统。bug报告虽然名字中只有“bug”，但是实际上它们并不只是用于追踪缺陷的，还用于追踪新功能

请求（在Bugzilla中叫做“改善”）。也就是说，每个报告都包含了对系统进行的一次逻辑修改的信息，而这次修改有可能是修复缺陷，也有可能是添加新功能。

表21-2展示了两种修改的比例。虽然Mylyn的修改有超过1/4是属于改善类，但对于这三个项目来说，大部分的修改还是修复系统缺陷。

表21-2 系统的修改类型对比

项 目	修改数量	缺陷数量（%）	改善数量（%）
Evolution	1939	1792（92.4%）	147（7.6%）
Firefox	11 710	11 198（95.6%）	512（4.4%）
Mylyn	3055	2247（73.6%）	808（26.4%）

这三个项目都使用了源代码控制系统，用记录下每次代码变更提交的方式对修改进行管理和跟踪。每次提交都可能包含对多个文件的修改，并附带各种元数据，如提交者的ID（通常是用户名或者邮件地址）、提交的时间和日期以及对修改的文字描述。

单个逻辑修改可能会对应源代码控制系统中的多个代码变更提交。开发人员可以将一个逻辑修改细分为更小的子任务，完成每个子任务并提交相应的代码变更，然后再进行下一个子任务。有时候，一个新功能可能会有多个开发人员参与开发，每个人都独立提交他们自己变更的部分。虽然这些代码变更都是分别提交到源代码储存库的，但是它们都属于同一个逻辑修改。

21

所以，我们把“修改”定义为一个bug报告中包含的所有代码变更提交。我们使用了不分大小写的规则表达式 `/bug #?(\d+)/` 来从代码变更提交的描述中提取bug的ID，然后再把所有的处在同一个bug ID下的变更提交组合起来。为了举例说明，请看看下面这三个描述。它们取自Firefox项目中的三次修改提交，三者组合起来形成一个逻辑修改。

> Bug 385423。重构textrun缓存，让所有的textrun客户端都使用同一个基于单词的全局缓存。把剔除问题字符（如换行符）的责任交给单词缓存。r=vlad,smontagu
>
> Bug 385423。停止显示零宽度空格（ZWSP）、段落分隔符（PSEP）及行分隔符（LSEP），并且不在平台textrun建立的时候传入这些字符。避免潜在的bug并强制执行统一的处理方案。r=vlad
>
> [OS/2] 解决gfxOS2Fonts.cpp无法编译的问题（模仿在修复Bug 385423时对gfxPangoFonts做出的修改）

我们的规则表达式没有匹配到的变更提交我们不予考虑。就Firefox和Mylyn来说，我们匹配到了大概3/4的变更提交，分别占总数的74.7%和75.8%。而对Evolution来说，我们却只匹配到了19.5%的变更提交。这是因为这个项目大部分的代码变更提交并没有明确提及bug ID，所以我们无法对这些提交进行分析。

由于我们的研究内容还包括弄明白修改的范围（尤其是单个逻辑修改所需要参考和修改的模块数量），所以我们还需要了解一下每个修改的大概规模。图21-3展示的是Firefox的代码修改规模（即每次修改过的代码行数）的分布情况，另外两个项目的分布情况也差不多。修改了超过1000

行代码的修改没有显示在柱状图中，以节省空间。表21-3的统计包含了全部的数据。

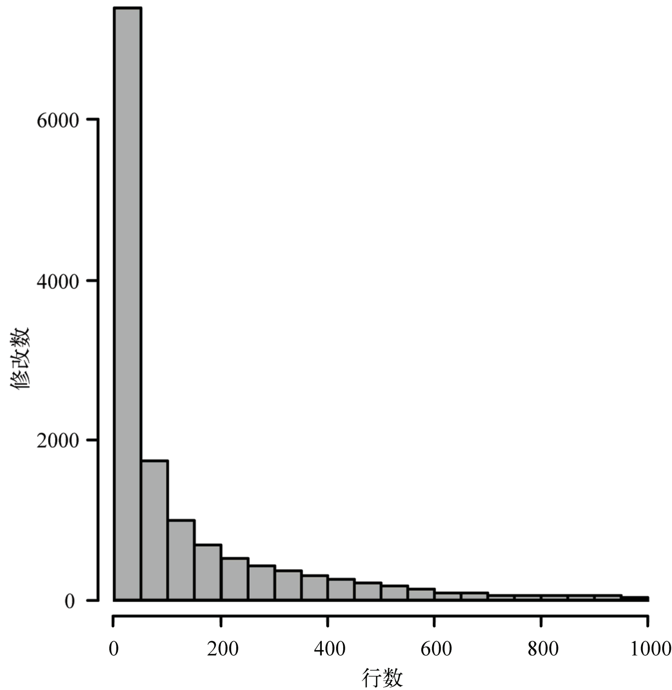


图21-3　Firefox的代码修改行数分布

表21-3　修改行数统计

项　　目	中　间　值	平　均　值	修改超过100行的百分比
Evolution	24	157.2	2.0%
Firefox	26	383.3	3.6%
Mylyn	62	248.5	8.6%

柱状图显示，这些系统中大部分的修改都非常小，而最右边有少许非常大的修改（如重构一个大型组件或者把代码从一个目录移到另一个目录）。

对于Myly项目来说，我们还可以通过大部分bug报告都附带的“任务环境”（task context）文件来找出更多的信息。正如我们前面简单提到过的，Mylyn是Eclipse IDE的一个面向任务的界面。它有一项功能是跟踪记录开发人员在进行任务时所交互过的所有软件部件。Mylyn对于进行任务时的一些“有趣”的代码元素进行了记录。这些元素不但包括修改过的属性和方法，甚至那些仅仅是在编辑器中浏览过的代码元素也包含在内[4]。然后开发人员就可以使用这些任务环境数据来过滤Eclipse中的视图，让它们只显示开发人员感兴趣的元素，让开发人员可以只关注和当前任务相关的那部分。

Mylyn项目有个政策，那就是解决bug的时候，必须先在相应的bug报告下建立新的任务。当开发人员完成任务之后，他需要将Mylyn收集到的任务环境信息附加在bug报告上。这样，其他开发人员就可以导入相关的任务环境信息到自己的Mylyn环境，并可以从解决问题的那个开发人员的角度来分析系统和修改。

图21-4展示的就是一个任务环境的例子。任务环境中加粗的元素由于和该任务密切相关，所以使用了更突出的格式来显示。图中左上角的滑动条代表元素的相关程度，开发人员可以滑动它来过滤想要查看的元素。开发人员们通常不会使用图中这种视图来进行工作，而是利用任务环境信息来过滤其他IDE视图，使其只显示相关的语言元素。图21-4中的视图主要是用于展示在附加到bug报告之前的背景信息。

每个任务环境数据中都包含了时间戳，可以用来估计开发人员在这个修改上大概花了多少时间。各种“交互事件”都有时间戳信息，包括选择事件（如当开发人员点击源代码中的元素的时候）、编辑事件（当开发人员修改编辑器中的文字的时候）以及命令事件（当开发人员调用IDE命令的时候）。可以预见的是，当一个开发人员正在进行工作的时候，会频繁地触发这些事件，生成相关的时间戳信息。也就是说，我们只需要将这些交互事件的时间戳按先后顺序排列，再计算两个连续的时间戳之间的时间差，并将这些时间差合计一下，就能知道修改所花的时间。为了避免把开发人员的中途休息时间也算进来，如果两个连续的时间戳的差别大于5分钟，我们就不予考虑。

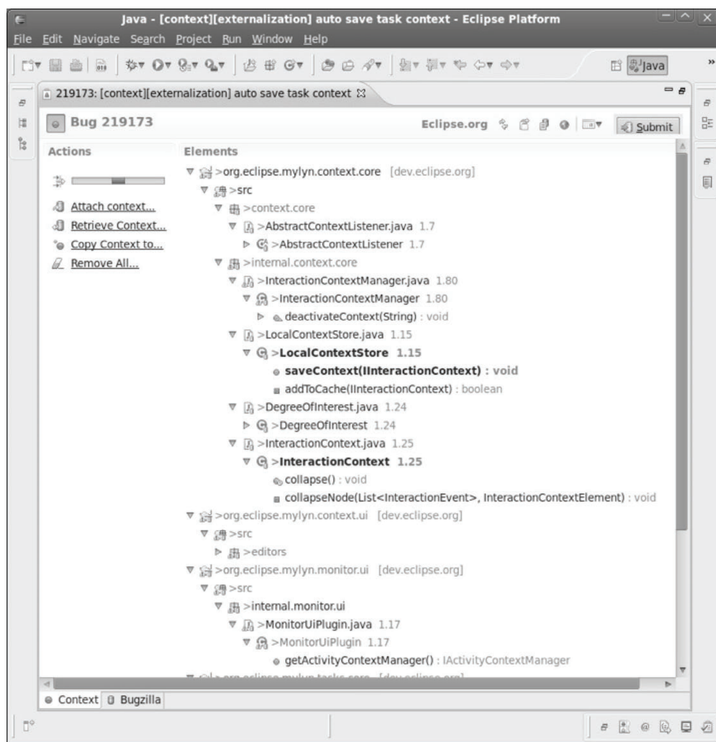


图21-4　Mylyn视图下的修改附加的任务环境

总而言之，当我们分析Mylyn的代码修改情况时，我们可以从两处获得数据：源代码控制系统中的提交记录，让我们可以得知开发人员对代码的修改；还有连接到bug报告的任务环境文件，让我们可以得知开发人员在做修改时所进行的步骤。从任务环境中储存的时间戳信息我们可以估计开发人员大概花在某个bug上的时间。这至少让我们可以对这个系统的修改情况进行更深入的分析，找出开发人员和系统的模块交互的更详细的信息，而不是仅仅去看最终的修改结果。

21.3　如何定义“模块”

要想讨论开发人员是如何和软件系统中的模块交互，我们就必须要对“模块”有一个统一的定义，并且弄清楚模块之间的代码是如何区分开来的。

虽然说我们可以直接用源代码中的声明来作为模块的定义标准，但是对于我们的研究来说，这样的选择面临两个问题。首先，我们选择研究的项目是用不同的语言来写成的，这些语言处理模块的方式各有不同。比如，Evolution是用C写成的，C语言基本不支持在源代码中定义模块。而Firefox主要使用C++，这种语言使用命名空间的方式来定义模块，但是Firefox本身并没有统一地使用这个语言特性来定义模块。此外，Firefox还包含了很多用其他语言写的代码，比如JavaScript和XUL（XML User Interface Language）等，都有着各自不同的模块化的方式。而Mylyn使用Java写成，是三个系统中唯一一个在源代码中明确定义了模块的系统，使用的方法是Java包。

其次，我们选择研究的几个系统都是足够大而且存在时间很久的，以至于如果我们太追究编程语言上的细节问题的话，我们的研究结果就可能就会受到与开发相关的问题（比如开发团队成员的替换或者代码的重构）的巨大影响。为了减少这些问题的影响，我们将采用稍微粗略一点的方式来关注模块的问题，即从架构的角度来看待模块。当然，如果能使用更为细化一些的模块定义来做后续研究应该会很有趣。

最后，我们没有使用编程语言的特性来作为模块的定义标准，而是使用了每个系统储存代码的目录结构。我们接下来即将证明，这些项目文件系统的组织方式可以作为一个通用的模块定义标准。此外，使用目录作为模块的定义标准还免除了我们区分不同模块的困难，因为不同模块就在不同的位置上。

Evolution和Firefox都使用了简单的目录结构，和我们在电脑上使用的结构差不多：每个顶级目录代表着一个主要系统组件，各个子目录把这个组件再分成几个稍小的子组件。下面是Firefox软件库的一瞥：

```
layout       network       parser
  /base        /base         /expat
  /generic     /cache        /htmlparser
  /inspector   /cookie       /xml
  …              …
```

每个子目录都可能包含更多的子目录，将源代码、测试文件和文档区分开来。

对于这样的目录结构，我们有两种备选的模块定义方案。往高了说，每个顶级目录（如layout）都可以被看做一个模块；往下一点说，每个顶级目录的子目录（如layout/base）都可以被看成是一个独立的模块。

Evolution是我们研究的三个项目中最小的一个，我们采取前一种定义。Firefox要比它大上几个数量级，所以我们采取后一种定义。如果使用前一种定义（即仅用顶级目录）来看Firefox，我们将得到巨大无比的模块，这个模块将包罗万象，让我们无法探知系统中各个模块之间的关联。而如果我们对Evolution使用子目录的定义方法，那么所得到的数据就会混杂不堪，使我们无法对其进行有效的分析。

Mylyn的目录结构和另外两个项目完全不同。Java的源文件是按照其所对应的包的声明来组织的，这样的结构盘根错节，比C/C++系统的目录层次要多很多。这些Java包最后又将组合形成Eclipse的子项目（即Eclipise IDE项目下的各个顶级目录），而这些项目本身也是按照同样的命名标准来组织的。下面是Mylyn的软件库一瞥：

```
org.eclipse.mylyn.bugzilla.core
  /src/org/eclipse/mylyn/internal/bugzilla/core
     /history
     /service
org.eclipse.mylyn.bugzilla.ui
  /src/org/eclipse/mylyn/internal/bugzilla/ui
    /action
    /editor
    /search
    ...
org.eclipse.mylyn.context.core
  /src/org/eclipse/mylyn/context
    /core
    /internal/core
```

和另外两个系统一样，针对这样的结构我们也有和此前一样的两个定义方法。代表着Eclipse子项目的顶级目录大致相当于Evolution和Firefox中顶级目录的子目录。如果再往上层说，我们还可以按照目录名字中的mylyn之后的第一个词来定义组件。比如说，org.eclipse.mylyn.bugzilla.core和org.eclipse.mylyn.bugzilla都可以被看做是更大的bugzilla组件的一部分。事实上，如果我们看一看Java包的结构，这两个目录都只包含属于org.eclipse.mylyn.bugzilla中的代码，所以这个选择也合乎语言的语义。

所以我们采取了Evolution的顶级目录定义法，而不是Firefox的子目录定义法，即采取org.eclipse.mylyn的直接子目录作为模块的定义标准。这样，我们就可以把存在于不同子目录中的核心逻辑和UI组件都看做同一模块的不同部分。

21.4 研究结果

我们已经对“修改”和“模块”做了具体而统一的定义，现在终于可以开始分析开发人员如何利用这些模块进行工作了。

21.4.1 修改的范围

我们的用一个简单的问题作为研究的开场：是否大部分的修改都被限定在了单个模块内？图21-5展示的是代码修改所影响到的模块数量，而表21-4是相关的总结统计。

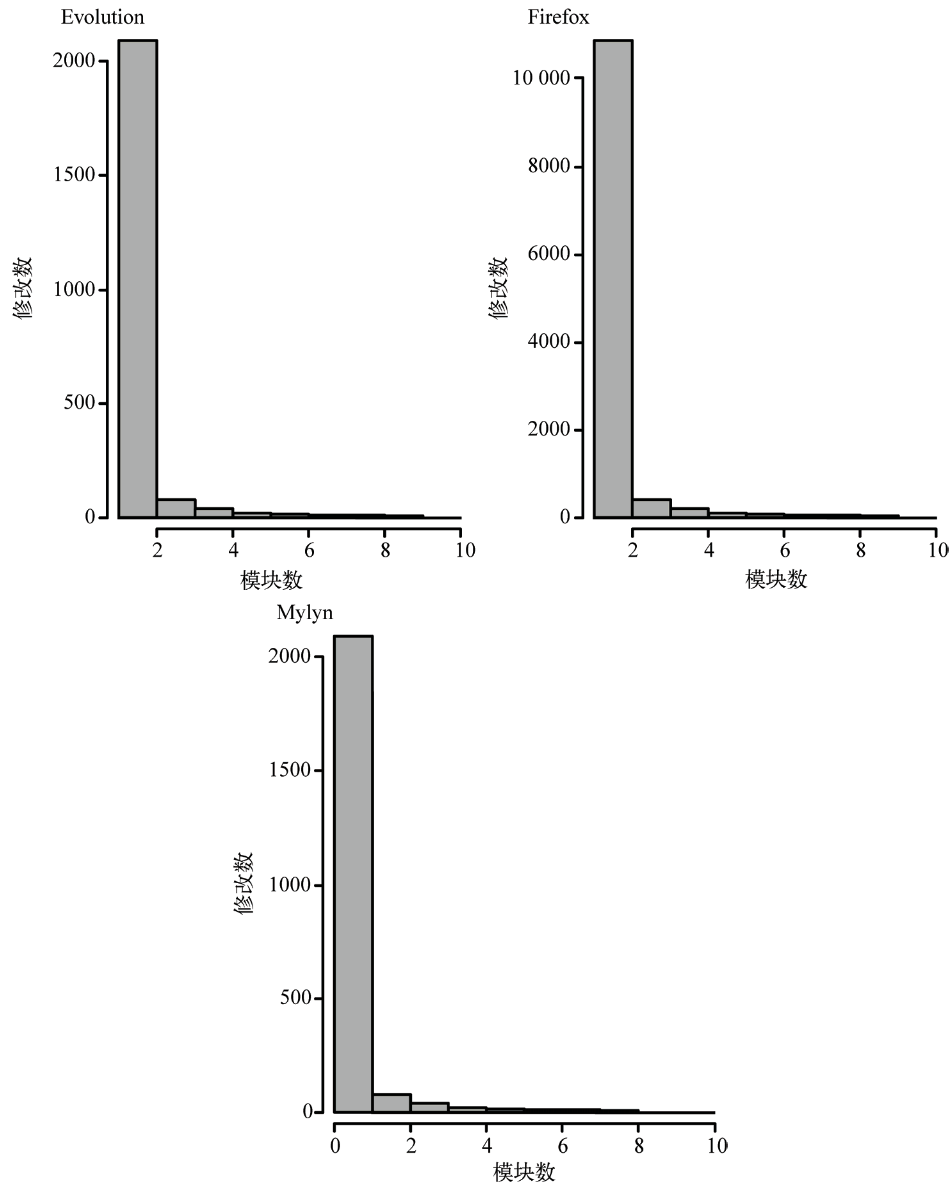


图21-5　每个修改在各个系统中所改变的模块数量

表21-4　修改影响到的模块数量

项　目	只影响了一个模块的百分比	平均影响模块的数量
Evolution	86.6%	1.243
Firefox	73.7%	1.577
Mylyn	69.7%	1.634

21.4.2 需要参考的模块

在修改代码的时候，开发人员必须参考多少模块？由于Mylyn有相关的数据，所以我们可以通过数字来回答这个问题，但是另外两个系统就不行了。

图21-6展示的是当开发人员做出修改的时候参考过的模块数量，平均数是2.365，中位数是2。参考过的模块平均数比实际修改的模块平均数（1.634）要高，也就是说，开发人员偶尔会需要参考一些他们最终不需要修改的模块。

如果我们再仔细看看参考过而又没有修改过的模块，就可以发现两个Mylyn模块非常突出。在13%的修改中，开发人员参考了Tasks模块但是没有修改它，而在8%的修改中，开发人员对Bugzilla模块也是如此。我们可能会推断，Tasks和Bugzilla相比其他的模块来说凝聚度较低，又或者说它们和其他很多模块有所耦合。我们稍候将用开发人员和系统的互动来定义模块，届时我们再来看这个推断是否正确。

那么，参考的模块数量对于完成任务所需的时间有着什么影响？我们的假设是，当开发人员参考的模块数量越多，完成任务所需的时间也就越多。我们的数据确实支持了这一假设：数据显示，参考的模块数量和任务所花费的时间之间有弱相关性（相关系数 $r=0.20$，$p<10^{-15}$）。

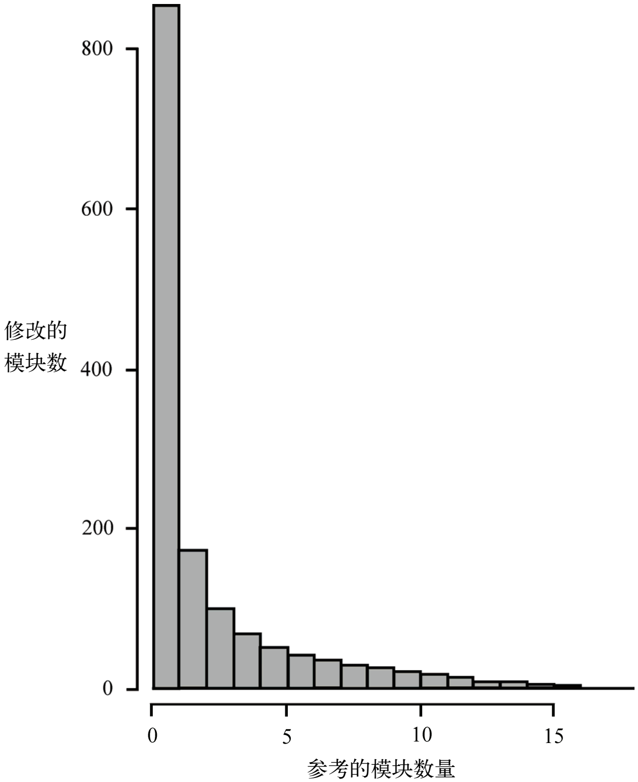


图21-6　Mylyn的每个修改所参考的模块数量

不同的开发人员所参考的模块数量是否也不同？特别是：经验更丰富的开发人员需要参考的模块是否更少？我们可以用开发人员关闭bug报告的数量来估计其对系统的了解和经验。将这个数据和每个修改中参考但未修改的模块数量相结合，我们就可以得出图21-7中表示的结果。

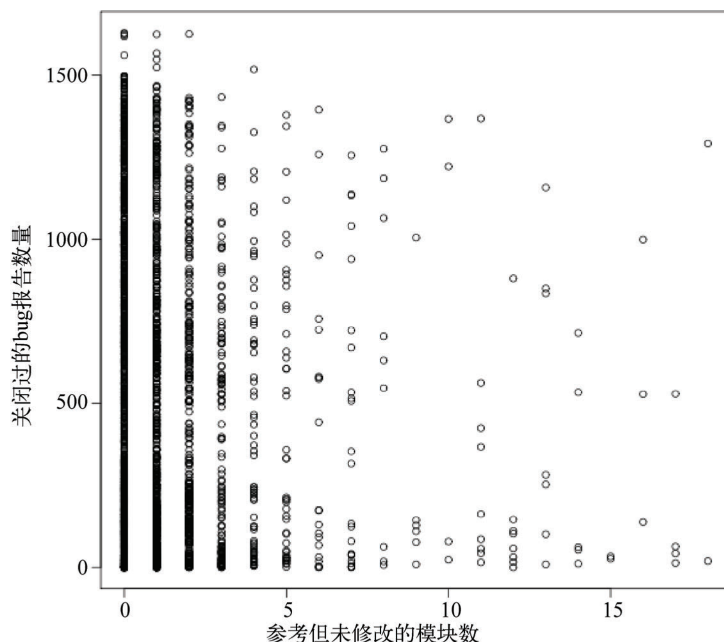


图21-7　开发人员的经验以及参考但未修改的模块数量对比。经验的衡量标准是开发人员在做出修改前所关闭的bug报告数量

我们在图的左边部分来说并没发现什么差别，新手和老手在做修改时都经常参考少量其他的模块。但是，在图的右边部分，情况就完全不一样了，需要参考很多其他模块的开发人员大都没什么经验。

21.4.3　自发式的模块化

到目前为止，我们的所有分析都是基于一个前提，那就是用这些系统的文件结构来区分模块是恰当的。我们现在换一个视角来看这个问题：能不能从开发人员的交互模式中推断出模块的定义呢？开发人员在工作时自发形成的模块分化与系统设计时明确定义的模块是否不同？前面的统计结果已经表明，对Evolution的修改几乎总是只改动一个模块（占总数的86.6%），所以我们这个阶段的研究将主要关注Firefox和Mylyn。

我们首先将同时修改过的模块配对，并对这些配对出现的频率做了简单的统计。对Firefox来说，出现频率至少在1%的配对分别是：

```
browser/base, browser/themes
browser/places, toolkit/places
```

```
layout/base, layout/generic
browser/themes, toolkit/themes
toolkit/locales, toolkit/mozapps
browser/base, browser/locales
toolkit/content, toolkit/themes
```

从这个列表中我们可以观察到几点。首先，顶级目录下的“base”模块看上去似乎和同目录下的其他的模块有着紧密的耦合。从设计的角度来看，这一点似乎并没有问题。其次，不同顶级目录下同样名称的模块也常常同时被修改：比如browser/places和toolkit/places常常同时被修改，同样的还有browser/themes和toolkit/themes。它们相似的命名表明系统架构师意识到有多种细分系统的方法，也表明可能会出现横切（crosscutting）的问题，而且选择任何一种模块化的方法都可能造成系统中高度凝聚的部分被分散到多个模块。而对于Firefox来说，将browser和toolkit分成两个组件就导致了places和themes组件的分散。

最有趣的是那些不符合这两种解释的组件关联。从前面的列表中我们可以看到，toolkit/locales和toolkit/mozapps似乎有着某种关联，同样的还有toolkit/content和toolkit/themes。对这些配对模块的修改细节的深入研究显示，系统的某些功能性其实可以提取出来，放到一个凝聚度更高一些的模块中去。

在对Mylyn采取同样的研究之后，我们发现了下面这些配对：

```
tasks, bugzilla
tasks, context
tasks, jira
tasks, help
tasks, commons
tasks, trac
```

我们很容易就能看出来：开发人员修改tasks模块时常常也会修改其他模块。我们可以回想一下之前的统计结果，在所有的修改中，有13%参考了tasks模块但是却没有修改它。将这个结果和我们观察到的配对情况相结合，我们就可以证明Mylyn的tasks模块和系统的其他部分是高度耦合的。

为了发掘tasks模块和系统其他部分的关系，我们采取了一种寻找数据之间的关联规律的数据挖掘方法，名为“多发模式挖掘法”（frequent pattern mining）[1]。我们用这种方法对Mylyn的任务环境数据进行挖掘，希望能发现开发人员在做修改时参考其他模块的某些规律①。在本次研究的环境中，我们得出的结果将以这样的形式出现：对于同一个修改，如果开发人员参考了模块A，那他就有可能会去参考模块B。找出的每条规律都有一个可信度，这个数值代表着这条规则所覆盖的数据集的百分比。我们对任务环境的数据挖掘得出了以下结果：

```
doc -> tasks (100%)
help -> tasks (100%)
trac -> tasks (84%)
team -> tasks (82%)
bugzilla -> tasks (81%)
jira -> tasks (81%)
```

① 关联规律是由ARtool实现的Apriori算法生成的（http://www.cs.umb.edu/~laur/ARtool）。

这些还只是可信度超过80%的规律。我们可以看出，开发人员在进行某个任务的时候，只要参考了箭头前面的6个模块（如doc，help等），就很有可能还需要参考tasks模块。

在确定了在Mylyn中的“条条大路通tasks”这个规律之后，我们可以再深入研究一下这些模块之间的联系是否处在“类”这个层面上。首先，我们对任务环境数据进行了挖掘（从类的角度），并提取了以下这些可信度超过60%的结果：

```
NewBugzillaTaskEditor -> AbstractRepositoryTaskEditor (88%)
BugzillaTaskEditor -> AbstractRepositoryTaskEditor (85%)
NewBugzillaTaskEditor -> BugzillaTaskEditor (79%)
IBugzillaConstants -> BugzillaClient (71%)
JiraRepositoryConnector -> AbstractRepositoryConnector (63%)
BugzillaRepositorySettingsPage -> AbstractRepositorySettingsPage (62%)
SynchronizeTaskJob -> RepositorySynchronizationManager (61%)
```

我们找出来的这些结果中大部分是具体实现类和抽象基础类之间的联系。开发人员需要浏览抽象类及其具体实现的子类，这看上去似乎是比较合理的，所以对这些关系我们并不感到意外。

如果我们改变策略，对修改数据进行数据挖掘，只考虑那些实际修改过而不是仅仅参考过的类，最后我们得到了截然不同的结果（仅列出可信度超过70%的结果）：

```
ITaskListExternalizer -> BugzillaTaskExternalizer (87%)
NewAttachmentWizard -> AbstractRepositoryTaskEditor (81%)
AbstractJavaRelationshipProvider -> XmlReferencesProvider (80%)
ActiveHierarchyView -> ActiveSearchView (76%)
PdeStructureBridge -> AntStructureBridge (75%)
RepositoryTaskAttribute -> AbstractRepositoryTaskEditor (74%)
ContextRetrieveAction -> ContextAttachAction (74%)
TaskCategory -> TaskListView (71%)
ContextAttachAction -> ContextRetrieveAction (71%)
```

这些结果让我们看到了系统的另外一面。比如，我们可以看到ActiveHierachyView和ActiveSearchView之间的紧密联系，同样的还有ContextRetrieveAction和ContextAttachAction。这些View之间，以及Action之间，可能都有着某种公用的功能性，而这些功能性也许应该被提取到独立的类里面。有些关联是出乎我们意料的。比如，我们不太明白PdeStructureBridge和AntStructureBridge为什么会有如此紧密的关联，因为虽然它们是从一个基本类派生出来的两个子类，但是两个类的实现却是大相径庭。

这样的结果表明，对于Mylyn这个项目来说，从同时修改的角度来做模块划分可能与用Java包和类来做的模块有很大差别。当两种划分模块的方式产生分歧的时候，即便只是做一个很小的逻辑修改，开发人员也可能需要更多地参考和顾及系统的其他部分。

21.5　有效性的问题

我们研究的基础是：使用bug报告来作为系统修改的逻辑单位。这个基础的有效性面临的第一个问题就是我们用了简单的模式匹配技术来关联bug报告和修改提交。使用模式匹配并不能保证可以找出bug报告和代码变更提交之间的所有关联：有一些变更提交描述中的bug ID使用了不

标准的格式，我们的规则表达式无法匹配到，而有的变更提交可能根本就没有明确提到bug ID。特别是Evolution，对于这个项目我们只匹配到差不多20%的变更提交，这使我们只能研究这个项目的一小部分历史，而无法窥探其全貌。

关于有效性的第二个问题来自Bugzilla软件仓库的干扰。我们认为每个bug报告都对应到一个单独的逻辑修改，但是事实并非如此。比如，有的报告描述的是日常软件维护（如升级第三方的软件库到最新版），而这样的报告和我们想要的研究报告（即缺陷修复或者功能改善）的特点完全不同。

这次研究的内部有效性的主要问题是我们的数据和得出的结论之间的关联。我们做出的很多结论都是对数据的主观诠释，所以如果有不同的结论也很正常。但我们可以用两点来量化我们结论的有效性。首先，我们的分析结果显示，参考的模块数量和修改所花的时间是紧密相关的，其皮尔森相关系数达到了99%。其次，对于从Mylyn的任务环境和修改数据中挖掘出来的每条关联规律都有对应的可信度，而我们只选择可信度较高的规律进行分析。

所以，我们的研究结果是有可能无法推广到其他完全不同的系统的。对于外部有效性来说，也有两个问题。首先，我们只分析了三个系统。虽然我们选择的三个系统在各个层面（如程序语言、规模、开发时间）都有所不同，但是如果我们想要将我们的研究结果概括推广，这样的几个系统还是太少。而且，这几个系统中只有一个（即Mylyn）有关于开发人员做修改时对其他模块进行参考的数据。其次，我们分析的这三个系统都是开源的，这就意味着修改的模式可能和商业系统不同。

21.6 总结

我们研究了三个不同规模的开源系统档案，目的是探索开发人员如何和这些系统所定义的模块来进行交互。虽然这些系统的规模、领域和编程语言都有所不同，但我们仍然发现了它们之间的一些共性。首先，我们如愿地观察到大部分的修改都被限定在了单个模块的范围内。但是，我们同时也发现了大量需要跨多个模块的修改，这让开发人员不得不参考系统的其他部分，并在进行修改的过程中花费更多的时间。其次，我们使用了开发人员的视角来看待这些修改，对任务环境信息进行了数据挖掘，找出了开发人员在进行修改时常常会参考或者同时修改的文件及类，并发现用这种方法来划分出来的模块和系统设计时划分的模块并不一定一致。

这些结果表明，虽然从目前看来，模块化方法对开发人员的工作应该是有好处的，但是仍然有改进的空间。比如说，如果有工具可以帮助开发人员查看同事在处理同类修改的时候参考过的模块，那么做修改所需要的时间就有可能降低。对开发人员和系统模块之间的互动的研究可能表明系统有重新模块化的需要，也有可能表明我们需要一些新的语言特性，以帮助开发人员更好地从多个角度来表示系统中的模块。

21.7 参考文献

[1] [Agrawal et al. 1993] Agrawal, R., T. Imielinski, and A.N. Swami. 1993. Mining association rules between sets

of items in large databases. *Proceedings of the 1993 ACM SIGMOD international conference on management of data*: 207-216.

[2] [Dijkstra 1968] Dijkstra, Edsger W. 1968. The structure of "THE"—Multiprogramming System. *Communications of the ACM* 11(3): 341-346.

[3] [Gosling et al. 1996] Gosling, James, Bill Joy, and Guy Steele. 1996. *The Java language specification*. Boston: Addison-Wesley.

[4] [Kersten et al. 2006] Kersten, Mik, and Gail C. Murphy. 2006. Using task context to improve programmer productivity. *Proceedings of the 14th ACM SIGSOFT international symposium on foundations of software engineering*: 1-11.

[5] [Kiczales et al. 1997] Kiczales, Gregor, John Irwin, John Lamping, Jean-Marc Loingtier, Christina Videira Lopes, Chris Maeda, and Anurag Mendhekar. 1997. Aspect-oriented programming. In *Lecture Notes in Computer Science, vol. 1241, Proceedings of ECOOP'97*, ed. M. Aksit and S. Matsuoka, 220-242. Berlin: Springer-Verlag.

[6] [Kiczales et al. 2001] Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An overview of AspectJ. In *Lecture Notes in Computer Science, vol. 2072, Proceedings of ECOOP 2001*, ed. J.L. Knudsen, 327-353. Berlin: Springer-Verlag.

[7] [Liskov et al. 1977] Liskov, Barbara, Alan Snyder, Russel Atkinson, and Craig Schaffert. 1977. Abstraction mechanisms in CLU. *Communications of the ACM* 20(8): 565-576.

[8] [OSGi 2007] OSGi Alliance. 2007. About the OSGi service platform: Technical Whitepaper. Available at *http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf*.

[9] [Parnas 1972] Parnas, David L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12): 1053-1058.

[10] [Tarr et al. 1999] Tarr, Peri L., Harold Ossher, William H. Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of the International Conference on Software Engineering*: 107-119.

第22章

设计模式的证据

Walter Tichy

设计模式是用于解决各种设计问题的可重复使用的解决方案。在1995年Erich Gamma等人[4]写了著名的《设计模式》一书之后，这个概念逐渐开始流行开来。在引入设计模式这个概念之前，程序员们在设计软件的架构的时候只能依靠少数几个一般设计准则。比如信息隐藏准则，即把软件分成数个模块，模块的接口保持不变。这条准则确保对模块内部的改动不会影响到接口，所以即便模块有了改动，使用模块的软件也无需跟着改动。这条准则是由Parnas在研究了改动频繁的软件之后总结出来的[8]。

而设计模式比起这些一般准则来说，又更进了一步：这些模式针对的是具体的设计问题。也就是说，设计模式就是设计好的“算法”。设计模式和算法都是解决具体问题的方法。比如，快速排序算法就是为了解决排序的问题，而观察者模式则是为了解决传递更新信息到软件组件的问题。一般来说，程序员们无法直接使用书本上的算法，因为它们通常是用伪语言来表达的。要想使用这些算法，程序员必须对它们进行调整、修改，并将这个算法对应到目标程序语言。在这一点上，设计模式也是一样，它们也不是设计定稿。实际上，它们所描述的只是通用的软件结构以及软件组件之间的交互，必须通过修改以适应具体情况。

设计模式的支持者们声称它有以下这些优点：

- 提高软件质量；
- 提升程序员效率；
- 促进团队沟通；
- 改善初级程序员的设计水平。

在设计模式刚刚出现的那些年头，没人知道设计模式是不是真的有这些优点。在那时，经验丰富的程序员们已经经历了很多昙花一现的编程潮流，所以不愿意再花时间和精力在新的潮流上。“设计方法”这一领域已经是人满为患了：这个领域包括了结构设计、功能设计、数据抽象化、模块化设计、面向对象的设计、多范式设计、架构风格、重构，等等。谁又会浪费时间在这种新推出又没经过验证的所谓设计模式上呢？不过万一这个模式是管用的呢？我们怎么来解决这个问题？应该听信相关书籍作者的言论、个人的经验，还是顾问的建议？这些渠道要么过于主观，要么就是基于个人经验的一家之谈。要想客观地验证设计模式的这些好处是否可信，只有一

个办法：客观地观察程序员的工作，更确切地说，就是进行一次科学的实验。这个实验将找出使用和不使用设计模式所造成的差别。如果设计模式真的能如声称的那样可以大幅提升效率和质量，那么我们应该能够观察到一些差别。由于设计模式所承诺的的优点太多，单独的一个实验将无法测试所有的优点，所以，在1996年，我和我的学生决定开始一系列的实验。在这个过程中，我们不但学到了很多关于设计模式的知识，还了解到了软件研究中的很多实验方法。在我们了解这些方法之前，不妨先来看看设计模式是什么，这将让我们大概了解其原理和意义。

22.1　设计模式的例子

在这一章中包括对下面7个设计模式（有时几个模式一起）所进行的实验：抽象工厂模式（Abstract Factory）、桥模式（Bridge）、组合模式（Compositum）、装饰者模式（Decorator）、观察者模式（Observer）、模板方法（Template Method）以及访问者模式（Visitor）。我们将详细了解一下观察者模式，其余的做简单介绍。如需完整而准确的模式规范说明，请参见参见文献[4]。

观察者模式解决这样一个设计问题。假设某个应用程序包含一个重要的数据结构，这个结构将会不断地更新。这个数据结构叫做*主体*（subject）。在主体变化的时候，需要通知另外一些组件。这些组件被称之为*观察者*（observer），因为它们时刻注意着主体的变化。有一点非常重要，那就是观察者的数量是未知的：它们独立运作，可以来去自如。这意味着主体和观察者无法被放到一个单独的类或者模块中去。也就是说，主体和观察者之间必须有一个动态的连接。对于这个设计问题的解决方案如下：主体和观察者都是独立的对象，如果观察者想要关注某个主体必须在该主体处注册。当主体变化时，它会传送一个消息或者提醒给所有注册过的观察者。我们可以把某个博客和对这个博客感兴趣的读者看做是观察者模式的一个例子。读者不必一直不停地刷新这个博客，而只需在博客处注册，并选择在新的博文发表时自动收到提醒（如邮件或者弹出信息），然后读者可以自行决定什么时候下载和阅读这篇新的文章。

图22-1展示的是观察者模式的类图。从图中可以看出，主体和观察者都有具体的子类，也就是说在使用的时候，我们必须根据使用环境对其进行调整。观察者类提供了三个重要的方法。`attach`和`detach`方法用于注册和注销观察者。`notify`方法将遍历调用每个已注册的观察者类的`update`方法。每个具体的观察者类都将实现自己版本的`update`方法。每个`update`方法都会从具体的主体类那里获取相关的数据，并更新所在观察者类的相关变量。这个模式有一个重要特点，那就是观察者之间相互独立（观察者之间无需传递数据），但是只要主体产生变化，那么所有注册过的观察者都将自动更新。

抽象工厂模式，又称为套件（kit），是一种用于创建多个同类对象的方法。我们可以用UI工具包中的元素作为例子，比如窗口、按钮、滚动条、图标等。在UI初始化之后，开发人员就必须选择符合当前平台的图形元素。要做到这一点，开发人员可以使用一个工厂类来生成需要的元素。如果平台变化，使用的工厂类也随着变化。所有这些工厂类都有一套相同的接口，这些接口由一个抽象类定义——这就是这个模式名字的来历。这个模式的好处就是：由于工厂类的接口是固定

的，所以UI只需要知道怎么选择合适的工厂类，这些工厂类就可以被分别独立实现，而无需考虑实际选择的是哪个工厂类。

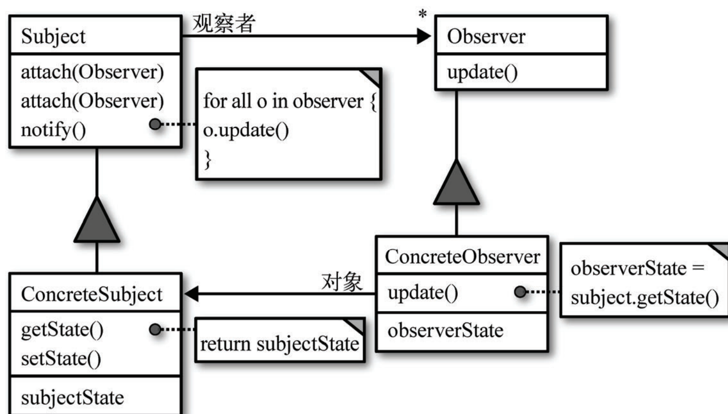


图22-1　观察者模式（选自参考文献[4]）

桥模式是另一种解决跨平台问题的模式。假设开发人员想要建立一套跨多个平台的UI工具包，而且他们想要从零开始编写这些UI元素，不用各个平台的原生UI元素。要想将这些元素画出来，就需要用到原始的绘图操作，比如绘制直线、绘制文字以及设置颜色等。在桥模式下，开发人员将把这些原始操作全部整理成一个接口，然后在多个平台上实现这个接口。当初始化UI工具包的时候，程序将选择适用于当前平台的原始操作版本，并在每个UI元素中都保存一个指向它的指针。在需要绘制按钮或者窗口的时候，所有的UI元素都会通过这个指针来调用原始绘图操作。这个指针就是一座“桥”，用于连接UI元素和针对不同平台的原始操作。

组合模式用于建立软件部件之间的层级结构。电脑的文件系统就是一个部件层级结构：根目录包含所有的文件和目录，根目录下的目录又包含其他的一些文件和目录，等等。部件层级机构可能延伸到任意的深度。组合模式提供了一个标准化的方法来安排这些层级结构。它把层级结构中的节点分为两种：容器节点（即我们例子中的目录），以及叶节点（即我们例子中的文件）。叶节点不可再分，也就是说它们无法包含其他节点。容器和叶节点的接口将提供一些两者都需要的公用操作。在文件系统的例子中，共同的操作有重命名、移动文件或目录以及打开文件或目录等。

装饰者模式的作用是在不修改已有类的情况下，为其附加信息或者功能。“装饰者”就是代替原来类的代理。只要装饰者类收到消息，它就会执行自己的函数，再把需要执行的工作委托给原来的类执行。例如，测试装饰者类可以在测试用例执行之前和之后附加代码，以记录测试的ID及其执行结果。

模板方法模式用于创建可扩展的软件。模板方法是指调用其他辅助函数（即所谓的hooks，“挂钩”）的大型函数。挂钩函数是抽象的，它们不会在定义模板的类中实现。模板方法类的子类必须实现这些挂钩函数。网页浏览器是一个很好的例子。浏览器提供了一个方法来播放媒体文件，但是这个方法必须可以扩展来支持其他厂商的未知格式。使用模板方法的话，浏览器就可以载入

媒体文件，但是使用挂钩函数来进行播放、停止、倒退操作。这些挂钩函数将由每个媒体类型对应的子类来实现。模板方法将处理媒体文件的所有组织层面的操作，但是当需要播放的时候，它就会将播放的任务交给适当的插件（即挂钩函数）来进行。

访问者模式提供了一套方法来遍历递归的数据结构。通常情况下，访问者模式和组合模式是一起使用的。例如，文件系统的访问者函数可以遍历整个目录树，以计算文件和目录所占用的空间。还有的访问者函数可能会从文件中提取关键词，并进行索引，以便快速查找包含某个关键词的文件。访问者模式的有趣之处在于，这些访问者函数并不是像人们想象的那样内置在文件系统的类中。实际上，访问者函数可以独立创建并随意用来遍历任何一个递归的数据结构。

22.2　为什么认为设计模式可行

在开始试验之前，我们应该先构想一个解释（或者理论），即为什么我们假定的现象可能会发生。如果我们毫无目的地比较方法A和B，也不知道他们之间为什么可能会有区别，就很难设计出令人信服的实验。从另一个方面来说，如果我们知道有哪些可能的原因，就可以顺着这个方向来设计我们的实验，否则就只能是大海捞针了。此外，普遍适用性也存在问题：我们只能宣称我们观察到的区别存在于和我们的实验环境类似的环境中。对于想要了解理论和实验之间的相互依赖关系的读者，我们强烈推荐Chalmers的书《科学是个什么玩意儿？》（*What Is This Thing Called Science*?）[1]。

那么，到底凭什么说设计模式可以改善程序员的效率、软件的质量以及团队的沟通？“区块理论”是一种可能的解释。1956年，George A. Miller在他的著名论文“神奇数字七，加减二：我们在信息处理容量上的一些限制”（The Magical Number Seven, Plus or Minus Two：Some Limits on our Capacity for Processing Information）[6]。他的理论假设人类的记忆包含短期和长期两种。短期记忆快速而准确，但是它的容量有限，只有大约7±2个“区块”。区块是信息的存储单位。有趣的是，每个区块内所存储的信息量却非常不固定。而长期记忆拥有巨大的容量，但是要想回忆起其中的信息相对较慢而且有时较为不准确。要想在长期记忆中存储信息，人类必须不断练习，以不断重复的方式来记忆。短期记忆无需练习；它似乎能毫不费力地存储信息，但是记得快忘得也快。在思考的时候，人类主要依赖短期记忆，但是当把所有短期记忆都用完之后，他们就会使用长期记忆，这样思维就会慢下来。比如，当面对大型软件的时候，程序员常常需要返回重新阅读或者检查他们已经解决的部分。当产生“怎么又忘了”这样的问题的时候，很明显短期记忆就已经被用完了，必须要再练习。短期记忆有些类似微处理器的缓存：如果新的信息放不进来，旧的信息就会被释放掉。但是，被释放掉的信息并不会自动储存在长期记忆中，除非它们经过了练习。

短期记忆的七个区块限制正是设计模式可以起作用的地方：这些模式可以将多个设计元素组合成一个存储单位。举例来说，如果不使用观察者模式，程序员需要在短期记忆中存储4个独立的类，加上它们之间注册和更新的协议，这样就用掉了不少记忆空间。但是，如果程序员使用观察者模式，那么他就只需要一个记忆区块，这样就腾出了不少记忆空间来解决其他的设计问题。

软件的设计常常会包括很多的类，所以这样把各个类放到一个区块中进行记忆的方法可能会起到一定的效果。程序的文档还可以直接写明使用的设计模式，并告知相应的类，这样就可以帮助读者把信息进行分区。比如，如果文档写明，使用的是观察者模式，程序员就可以很快地对这些类有一个大概的认识，甚至不需要专门去看注册和更新的代码。从本质上说，设计模式利用了更多的程序员的短期记忆，使得存储更完整，思考的过程更快、更全面也更准确。这也正是设计模式宣称能提高程序员效率和软件质量的原因。团队的沟通也可能得到改善。举例来说，在使用设计模式之后，程序员们就不必长篇大论地告诉其他人哪几个对象如何注册以及获知更新等，而是只需要说哪几个类组成了观察者模式就可以了，无需多言。看来，专门的术语确实有它的优势啊！

不过这些还只是推测，没人知道设计模式到底能不能真的带来这么多好处，但是至少区块理论可以帮助我们设计合适的实验。这些实验将用于测试程序员在使用设计模式中的类和各类软件元素时的效率。实验中的任务必须足够大，这样才能让程序员们无法完全利用短期记忆来工作，但是也不能过于庞大。我们认为设计模式的好处主要可能来自于以下两点：设计模式可以让程序员更准确地了解软件的结构，或者设计模式帮助开发者腾出了一些短期记忆的空间来做其他事情。由于短期记忆的实际使用量难以确定，所以我们的实验将主要关注前一点。

22.3 第一个实验：关于设计模式文档的测试

我们需要回答的第一个问题是：仅仅使用设计模式文档是否就能提高程序员的效率？我们的想法是，让两组程序员对同一个程序进行修改，其中一组的文档中包含现在使用的设计模式信息，而另一组则使用没有设计模式信息的普通文档。也就是说，这个实验测试的是设计模式文档的影响，而不是模式本身的效果。我们在1996年设计这个实验的时候选择了这种方式，因为当时我们尚未找出一种方法可以为一个程序设计两个相同而又具备可比性的版本，即一个包含设计模式，另一个不包含。不过在后来的一次实验中我们解决了这个问题。无论如何，就算只测试设计模式文档也还是有意义的，因为它可以让我们初步确认设计模式是否可能有效。如果单单在文档中包含设计模式信息就能够提升效率，那么可以想象，使用和不使用设计模式的程序之间的区别会更大。

22.3.1 实验的设计

我们想要用实验来回答的具体问题是这样的：相比起没有任何设计模式信息的代码，包含了明确的设计模式注释的代码是否更能够帮助开发人员进行维护工作？我们把这个问题分为两个假说。

- 假说1

  设计模式文档加快了其所对应的维护任务的进度
- 假说2

  设计模式文档减少了其所对应的维护任务中的错误

实验将记录程序员在执行设计模式所对应的维护任务时所花的时间和发生的错误。实际上，

我们进行了两次实验。第一次实验是1997年1月在德国卡尔斯鲁厄大学（下称UKA）进行的，而第二次实验是1997年5月在圣路易斯华盛顿大学（下称WUSTL）进行的。在UKA，64个研究生和10个本科生参与了实验；在圣路易斯，22个本科生参与了实验。所有的学生都参加了Java（UKA）或者C++（WUSTL）的培训，并在实验之前都已经使用设计模式写过程序。我们在正式实验之前让所有参与者先参加了一次测试，以确保他们了解实验相关的设计模式。详细的过程请参见参考文献[10]。

常常有人会提出异议，说我们应该找专业人士而不是学生来参与实验。但是，在那个时候设计模式刚刚起步不久，有相关经验的专业人士非常难找。不过，即便实验的对象是学生，我们的实验也仍然是有效的。如果用不用设计模式对学生来说没什么区别，那么对于专业人士来说设计模式估计也不能带来什么好处。造成这种情况的原因有几个：首先，专业人士有着与大型系统打交道的经验，所以他们可能不太需要设计模式的帮助，这样就会降低设计模式的真实效果，让使用和使用设计模式的区别不那么明显；其次，专业人士的背景更复杂，他们中很多并没有经过计算机科学专业的正规培训，而且他们的经验也从几年到几十年不等。与此相反，在大学学习计算机科学的学生大都同时阅读同样的材料，而且他们的编程经验也比较一致。也就是说，如果用专业人士作为实验对象，那么数据中的干扰将会非常多，而数据干扰的增多加上效果的降低就意味着我们将难以发觉那些不那么明显的差别。如果是这样的话，我们就不能用专业人士作为实验对象，因为结果将是不确定的。我们对学生做的实验可以被看做是一个初步实验，研究人员可以据此来判断假说在专业人士身上推广的可能性。

虽然两次实验是相似的，但同时也有一些小小的不同，这使得它们相互之间更为互补。在UKA，学生们使用Java语言来把答案写在纸上，而在WUSTL的学生们用C++来在电脑上写可以运行的程序。在纸上写答案可以避免很多和实验问题无关的影响，比如使用编程环境或者程序语言的困难等。而在电脑上写可以运行的程序可以更确切地证明程序员们真正理解了设计模式。对两次实验结果的比较让我们得出了一个结论，那就是在以后的实验中只用在纸上写的方式其实就足够了。此外，使用两种不同的面向对象的语言让我们可以说我们的结论并不只是基于某一种语言。

这两次实验都在期末的时候进行，代替期末考试。实验的长度在2到4小时之间，但是很多WUSTL的学生提前放弃了，因为他们想要在期末考试之后乘车回家。科学实验也是在现实世界中进行的，而现实世界的事情往往会对实验产生计划之外的影响，WUSTL的例子就是由于对此认识不足而付出的代价。

由于我们预先已经知道只有为数不多的参与者，我们决定让每个参与者先使用带有设计模式的文档进行一次任务，再使用不带文档的进行一次。这样，我们就能从每个参与者身上得到两个数据。显然，让参与者重复执行同样的维护任务两次的做法不是很好，所以，我们需要两个在规模和复杂度上都差不多的程序。由于我们要将两个程序的实验数据结合起来，它们的复杂度不能差别太远，因为如果差得太远那么我们就很难分辨区别到底是由设计模式造成的，还是规模和复杂度的不同造成的。第一个程序是电话簿。这个程序包含一个地址数据库，然后可以按多种格式来显示姓名和电话号码。它由11个类和565行代码组成，其中197行是注释。电话簿使用了观察者和模板方法两种设计模式。第二个程序实现了一个与或树（And-Or-Tree），一种递归的数据结构。

它包括7个类和362行代码，其中133行是注释。这个程序较短一些，但是也更难理解。它使用了组合模式和访问者两个设计模式。为了体现设计模式信息，我们在电话簿程序的代码中增加了14行注释，在与或树的代码中增加了18行代码。（此处的行数是Java版本的行数，C++版本的行数稍有不同。）下面是我们添加的注释的例子：

```
*** 设计模式：*** 注册两个 TupleDisplays 为 Tupleset 的观察者。
*** 设计模式：*** newTuple 和它的辅助方法 mergeIn() 组成一个 *** 模板方法 ***。挂钩方法分别是 select()、format() 和 compare()。
```

需要注意的是，即便没有这些设计模式信息，这些程序也还是有完备的文档的。就算没有这些额外的提示信息，维护任务也是完全可行的。也就是说，实验的设计极为保守，把设计模式信息的影响减到低得不能再低的程度。如果我们能用这样的实验条件证明设计模式可以减少错误或者工作时间的话，那么这些改善在真实的程序中可能会更明显，因为那些所谓“专业”的程序往往没有这么完备的文档。

对于电话簿来说，维护任务包括声明并实例化两个新的观察者类，一个有模板方法，一个没有。对于与或树来说，参与者需要声明并实例化一个新的访问者类。维护任务的说明并没有提到设计模式。

由于两个程序是不同的，所以我们必须解决研究的有效性的问题。如果在第一次实验中得到了模式信息的参与者在第二次实验中自己去寻找模式怎么办？那么第二次实验的数据就无效了，因为如果参与者自己去寻找模式，那么他们的行为就会和有模式信息时一样。对电话簿程序来说也是一样，参与者有可能从中了解到一些模式信息，然后应用到另一个的程序中。这种有效性的问题被称为顺序问题或者学习效应。我们解决这个问题的办法是使用制衡性实验设计：我们将参与者分为四组，每组收到任务的顺序，以及这些任务是否带设计模式文档的顺序都不同。

图22-2展示的是制衡性实验设计的详情。比如说，第1组先处理有模式文档的电话簿程序，然后再处理没有模式文档的与或树程序。通过交叉对比结果，我们就可以检查模式文档是否造成了起步上的差距。比如，第1组和第4组都处理了带模式文档的电话簿程序，但是先后顺序不同；第2组和第3组也用同样的方式处理了与或树。研究人员将检查左边圆圈（即有模式文档）和右边圆圈的实验结果的区别。如果确有很大不同，那么我们就可以说存在学习效应。同样的，我们也可以对没有模式文档的菱形也做同样的比较，找出它们之间是否也有学习效应。在我们的实验中没有发现学习效应。（为了更安心，我们又在实验后的调查问卷中问参与者有没有自己去找设计模式，结果是大家都没有。）由于没有发现学习效应，那么我们就可以直接对比圆圈和菱形的实验结果，而无需对学习效应进行修正。

22.3.2 研究结果

我们对参与者们提交上来的答案评了分，并计算了分配和完成任务之间的时间（精确到分钟）。此外，我们还计算了完全正确的答案的数量。我们发现评分并没有表现出明显的差异，所以只对比了时间和完全正确的答案的数量。下面这个表展示了与或树的结果。

	有模式文档	无模式文档	显著性（p值）
UKA，与或树			
正确答案	15 / 38	7 / 36	0.077
花费时间，平均值	58.0	52.2	0.094
7个最佳答案的花费时间，平均值	38.6	45.4	0.13
WUSTL，与或树			
正确答案	4 / 8	3 / 8	
花费时间，平均值	52.1	67.5	0.046

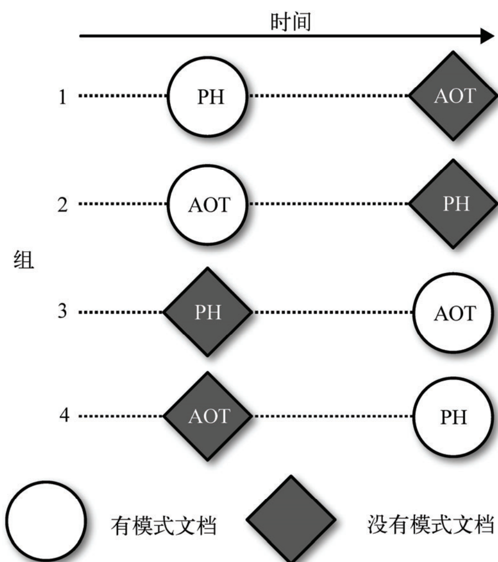


图22-2　模式文档实验的制衡设计（PH=电话簿，AOT=与或树）

关于这个表，有几点需要注意。首先，在设计模式文档的帮助下，UKA的参与者们提交的完全正确的答案数量增加了一倍（15比7，总数36），这个效果很可观。WUSTL的差距稍小一些。让人意外的是，有了模式文档之后UKA的学生平均花费的时间居然更长了（58比52分钟）。不过，这个看法有误导的成分，因为没有模式文档信息的那一组的正确率要低很多。这里需要注意的是UKA的答案是写在纸上的。在真实的维护工作中，错误的答案可能会被检查到并被改正，这样就会需要更多的时间。在纸上写的方式让参与者们很难检查自己的答案，所以我们在这次实验中并没有观察到多花的时间。很明显，对比在错误的事情上花的时间和在正确的事情上花的时间是没有意义的。所以，我们把实验数据精简了：由于没有模式文档信息的小组只提交了7个正确的答案，我们就把这7个正确答案所花的时间和对照组中最好的7个答案所花的时间进行对比。结果是有模式文档信息的组花的时间少一些，不过显著性不高。WUSTL组时间差别的p值小于0.05，这可能是因为这一组不只设计，而且还测试并改正了他们的答案，所以结果更平均一些。通过比较

工作时间和答案质量，我们还发现，在没有模式文档的情况下，速度较慢（能力稍差）一些的参与者提交的答案质量非常低，而对于有模式文档的小组来说，质量和所需的时间没有关联。

以上关于时间和正确度的讨论让实验设计中的一个缺陷浮现了出来，那就是没有考虑到质量和效率具有相互依赖性。很明显，要提高质量需要时间，所以花费更多的时间并不意味着设计模式没有价值。也就是说，只有当答案的质量相同时再比较时间才有意义。我们解决这个问题的方法是只对比正确的答案，但是这就让我们损失了一半的数据和统计显著性。吃一堑长一智，其实我们可以在提交时做一次验收测试，确保所有答案的正确性，就能解决这个问题。关于这个技巧我们将在结论部分再做介绍。

总的来说，与或树代码中的模式文档节省了一部分维护时间，并促成了更好的解决方案，甚至能力稍差一些的程序员也可以写出较好的解决方案。而对于电话簿来说，结果（未在本文中展示）也表明模式文档可以节省时间，但是没有关于质量的结果，因为数据不足（部分学生走掉了）。

我们的结论是如果需要维护的程序使用了设计模式，那么在代码中包含模式相关的文档就有可能减少执行程序修改所需的时间，而且还有可能改善修改的质量。所以，我们建议在源代码中一定要加入设计模式信息。如前所述，我们这次实验并没有测试有没有设计模式的区别，只测试了有没有模式文档的区别。不过，实验结果令人兴奋，所以我们开始准备第二次实验。这次，我们将回答设计模式有没有用这个问题，并使用专业人士作为我们的实验对象。

22.4 第二个实验：基于设计模式的解决方案和简单解决方案的对比

第二个实验是以一种出人意料的方式开始的。1997年的时候，sd&m公司在德国法兰克福附近举办了一次技术会议，我是受邀发表演讲的人之一。我的演讲主题是设计模式，包括我们早前关于模式文档的实验结果，以及测试设计模式实际效果的重要性。最后，我谈到正在为新的实验寻找参与者，作为参与的回报，我将提供设计模式的强化课程，并介绍当时已知的主要设计模式。此时，sd&m公司的创始人兼总裁Ernst Denert正好坐在前排。我说完之后，他站了起来，走到讲台上，并向我要麦克风。然后怎样呢？演讲之后确实是有问答时间，但是也不能抢演讲者话筒吧。然后，他转向了他的员工（在场的大概有150人，大部分是开发人员），说道："你们都听到Tichy教授说了什么。我没什么补充的，就问问，谁想参加？"几秒钟的沉默之后，突然很多手举了起来！我从来没见过更有效的实验动员了，从来没有。在经过了各种准备并处理了时间安排上的冲突之后，共有29名来自sd&m的专业开发人员参与了我们的实验。1997年11月，实验在sd&m的慕尼黑办公室进行。

我们希望用实证研究的方式确定设计模式是否能带来好处，以及是不是模式也有好用难用之分。尤其是，我们曾经发现有的开发人员对于设计模式过于狂热，以至于在使用简单的方法就能解决问题的时候他们也会使用设计模式。这样的矫枉过正会不会造成问题？按大脑区块理论来说的话，就不会，因为单个区块的容量并不重要。

不过，只有让程序员们修改两个程序，一个使用设计模式而另一个不用，再做对比，才能找出真正的答案。

和之前一样，这个实验也是调查在不同设计模式下的软件维护工作。不过这次我们的实验对象是专业人士。这次实验的结构和前一个类似，只是在强化课程前后分别进行了两次实验，共花了两个整天。在第1天早上，我们进行了一次课程前的实验，希望能评估在不了解设计模式的情况下，开发人员的效率如何。在第1天下午和第2天早上，开发人员们参加了一次设计模式的强化课程，然后第2天下午进行了一次学习后的实验。参与者包括29个专业程序员，工作的时间平均是4.1年。参与者们仍然使用纸笔，虽然不限时间，但他们都在三小时内提交了解决方案。实验中他们需要对4个程序进行维护，每个程序都有两个版本：一个设计模式版，包含一到两个设计模式，以及另一个功能相当的版本，使用了比较简单的结构。开发人员在两次实验中分别对设计模式版和普通版进行维护。两次实验分别在培训前后进行，这样我们就能知道关于设计模式的知识会不会提升效率，也可以看出哪些模式在没有专门学习过的情况下比较容易掌握。

图22-3概括了实验的设计。为了解决学习效应的问题以及疲劳带来的影响，我们调整了程序的顺序（即采取了制衡性设计）。Pat代表模式版本，Alt代表普通版本。ST、GR、CO、BO代表各个程序。F代表熟悉期，在这个时间段我们向参与者介绍实验的流程。

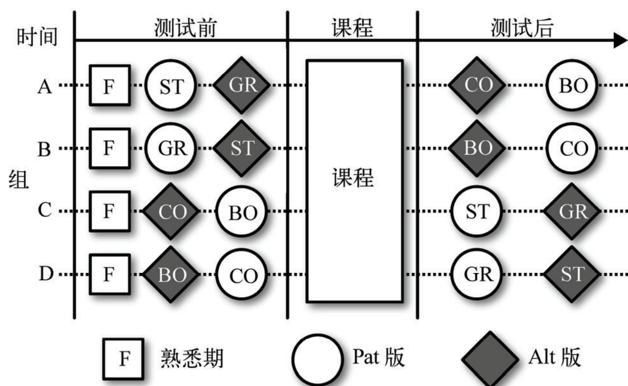


图22-3 实验设计：圆圈代表设计模式版，深色的菱形代表普通版本。双字母代码是程序名的缩写。时间从左到右进行

不同的模式得出了不同的结果。图22-4展示的是参与者们在维护ST程序（使用观察者模式）时所用的时间。4个盒状图展示的是修改时间的分布情况。有50%的数据处于沙漏形状之内，数据的中间值位于沙漏的中央。左边是学习之前的测试结果。此时，修改设计模式版比修改普通版本（“模块版”）花费了更多的时间。显然，未知的模式带来了额外的复杂性，让开发人员花费了更多的时间。不过，在经过了设计模式培训之后，情况就有了改变。普通版所花费的时间和培训前大致持平，虽然实际上设计模式的版本更复杂，但是花费时间却更少。这种情况正好符合我们根据大脑区块理论所做的设想。

图22-5展示的是装饰者模式的相关时间消耗。很明显，使用设计模式节省了不少的时间。然而，还有一个惊喜：培训前和培训后的结果没有区别！显然即便是在程序员没有专门进行学习的情况下，装饰者模式也很容易掌握和修改。也就是说，参与者们无需预先将装饰者模式放入记忆区块中。

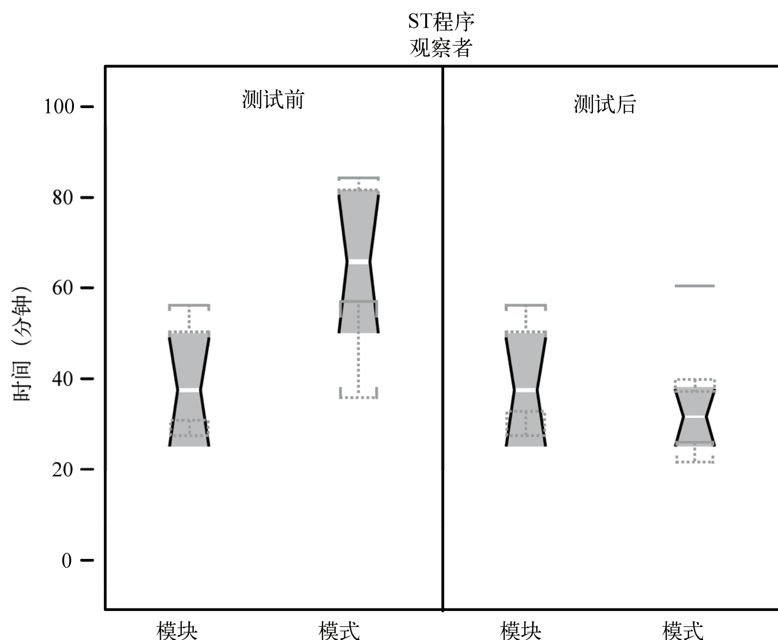


图22-4 观察者模式的时间花费对比

不过，并不是所有的实验结果都表明了这一点。例如，组合模式和抽象工厂的两个版本在培训前后的实验结果都没什么区别，而访问者模式给参与者们带来了很大的困扰。在培训前的实验中，普通版需要的时间比访问者模式版的要多，但是在培训之后，情况却反了过来。很明显，在培训课程中讲解访问者模式的时间相对较少，所以参与者们没能完全理解。总的来说，一共有9个维护任务。在大多数的任务中，我们都发现了设计模式的积极效果：要么因为设计模式所具备的灵活性免除了额外的维护时间，要么维护的时间相对于普通版本有所减少。在少数几个任务中我们发现了负面的效果：普通版本的解决方案较不容易出错，或者需要的维护时间更少。换句话说，不是所有的设计模式都是同样地好。如果想获取各个维护任务的详细信息，请参见参考文献[8]。

2002年，我们在挪威的Simula实验室扩展并重复了这次实验。我们使用了同样的程序和维护任务，甚至连培训课程和材料都完全相同。但是，不使用纸笔，而是使用实际的编程环境让我们的实验更加真实。共雇佣了44位来自各个大型IT咨询公司的专业人士参与了这次实验。我们用回归模型分析了消耗的时间和答案的正确性。我们还在线记录了参与者的工作情况，让我们可以更好地了解这些结果。数据非常清楚地表明了设计模式有好用和不好用之分。观察者和装饰者模式非常易于掌握和使用，甚至对于几乎没什么模式知识的参与者来说也是如此。访问者模式再次造成了困扰。这次实验证实了早前实验的结果。这一点非常重要，因为这次我们不是在纸上谈兵，而是实际进行了编程。此外，参与者的专业背景差别巨大（从1年以下到20年以上），所以这次的结果更具备普遍性。

这些结果的意义是：我们不能简单地说设计模式是好还是坏，还必须考虑模式是否适合想要解决的问题，以及程序员是否能理解这个模式。当程序的代码中包含了模式的信息时，就算是短

期的培训也可以提升维护工作的效率和质量。当我们在犹豫是否选择设计模式的时候，不妨考虑一下软件开发中随时可能出现的新需求，那时设计模式所具备的高灵活性就可以帮上忙。

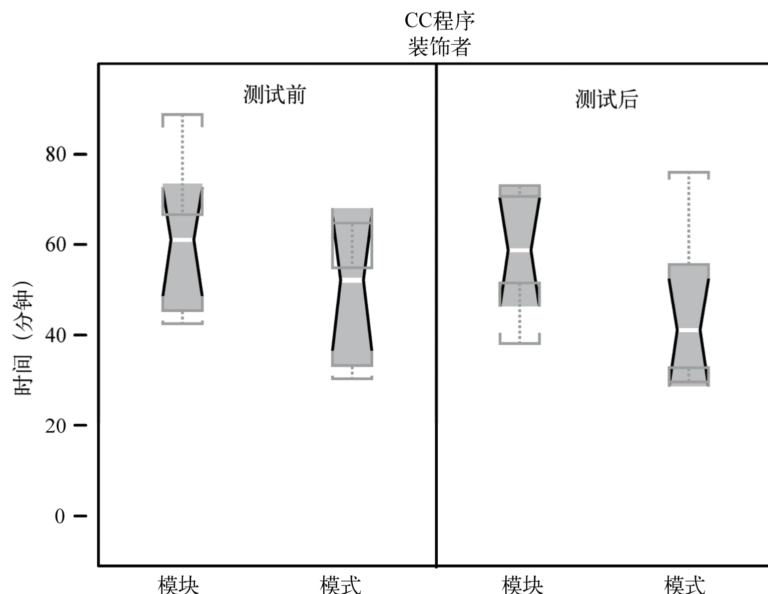


图22-5　装饰者模式的相关结果

22.5　第三个试验：设计模式之于团队沟通

关于设计模式的第三个大的假设是它能改善团队的沟通。我们第三次实验的目标，是测试一般性的模式知识能不能为团队成员带来更有效的沟通。很明显，要想检验这个假设是否正确，除了直接观察团队成员之间的沟通别无他法。难点在于如何让他们沟通，以及如何评估沟通的成效。自言自语或者直接和观察员谈话对于团队成员来说是不自然的，所以我们决定将参与者们分为一个个的二人小组，并在他们执行设计维护任务的时候记录他们之间的交流（包括音频和视频），然后再分析这些录音和录像。

实验的设置如下。每一组参与者都将执行一个设计维护任务。我们会预先给其中一个参与者额外的一些时间，作为诱导解释阶段的准备。在这段时间内，该参与者将研究需求以及程序的设计文档。我们称这个组员为专家。我们没有限制准备的时间，不过所有的参与者都只用了差不多1小时。在准备完毕后，第二个组员加入，我们称之为新手。专家将向新手介绍程序的设计。随后，两个组员将合作完成两个维护任务。这样的方法让我们可以观察到两个有趣的沟通阶段：一是专家给新手介绍程序的阶段，二是两个组员之间的合作阶段。

设计模式知识作为独立变量出现，也就是说，他们的设计模式知识都相同。首先，我们会在参与者们参加培训前做一次实验，然后他们会进行三个月关于设计模式的的培训，最后再进行一

次培训后的实验。整个实验从1999年4月一直进行到了同年8月。培训课程包含关于所有实验中涉及的（及没涉及的）各种模式的实用练习。培训前后的两次实验分别使用了两个不同的应用程序。同此前一样，为了平衡应用程序复杂度的区别，这些小组被分成了两批，并交替着进行实验。为了平衡个人能力的影响，在两次实验中，专家和新手的角色也会相互转换。15位参与者都是计算机科学专业的硕士生，都是第4学年在读，在实验之前没有设计模式方面的经验。

接下来的问题是如何分析录音和录像。其中的一个研究人员把交流的音频和视频转录成了文字对话，然后用了一个简单的方案来为这些对话分类。（这样的做法工程浩大。仅为一组的对话进行分类就花了4个多星期，而总共有12组对话。）分类方案将标出说话者，以及言论的类型，如提问、回答、反馈以及介绍设计时所做出的的宣言。介绍设计时做出的宣言尤为重要，因为其包含了关键的设计信息。分析的方法如下：每次专家的宣言我们都算作+1，每次新手的宣言算作－1。我们会计算出70条数据的移动平均值，并缩放到－10至10这个范围内。移动平均值的图表我们称之为沟通线。图22-6展示的是我们假设的理想沟通线。在介绍阶段，当专家介绍程序的时候，沟通线应该达到+10。在第一个任务分配完成后，如果新手和专家都同时参与讨论的话，这条线应该下降到0。两项任务分配的时间由两根垂直的实线表示；虚线表示移动平均值在实线附近的范围。由于沟通逐步趋于平衡，所以从10到0的下降不是垂直而是倾斜的。当我们把理想的交流线和实际的交流线相叠加的时候，就可以从两者之间的偏差中看出一些端倪，比如某个组员在交流中占了主导地位。通过对同一个小组在培训前后两次进行的实验的交流线进行对比，我们可以从质量和数量上看出，专家对程序的解释是否有效，以及合作阶段的沟通是否均衡。

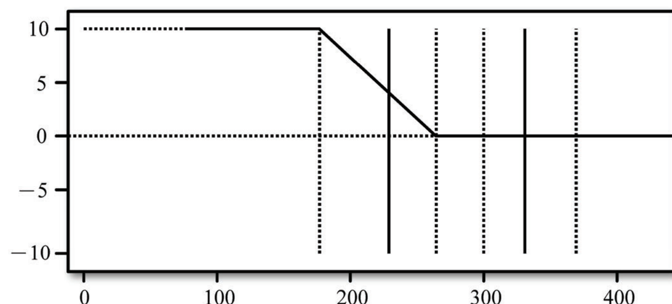


图22-6　理想的交流线

结果表明，在培训前的实验中，所有的交流线都严重失衡，而大部分培训后实验的交流线在解释阶段会有些“隆起”，然后逐渐均衡在0周围。图22-7展示了T1组的培训前实验结果。这里我们看到了新手占交流主导权的情况：专家开始解释，但是几乎就在同时，新手（刚刚开始对软件一无所知）就主导了讨论，这可以从交流线掉到0以下看出。我们再来对比一下图22-8中的交流线，这张图展示的是经过培训之后的情况。现在专家和新手互换了。请回忆一下图22-7之中主导交流的那个人（注意此时角色已转换）。现在，当他们讨论维护任务的时候，早前交流稍差一些的那个组员也能积极参与进来，二人的交流变得平衡起来。

图22-9和图22-10展示的则是一个专家主导交流的例子。在培训前的实验中，专家从头到尾地

主导了整个对话，但是在培训后交流变平衡了。

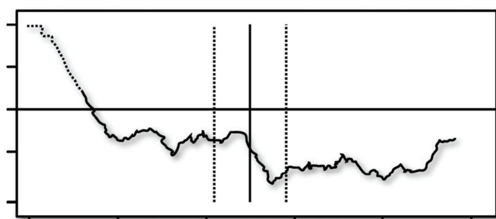


图22-7　T1组的交流线（培训前）

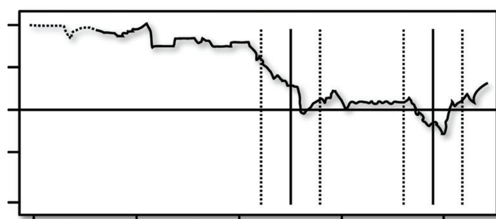


图22-8　T1组的交流线（培训后）

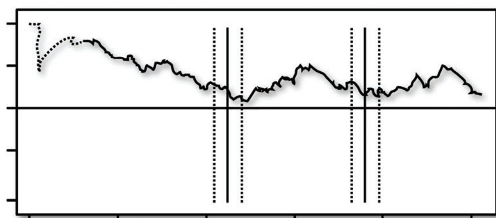


图22-9　T4组的交流线（培训前）

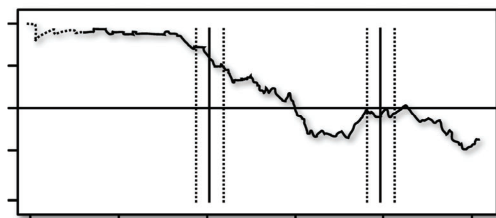


图22-10　T4组的交流线（培训后）

为了进行量化分析，我们计算了每个小组理想交流线和实际交流线的区别（包括培训前后两次实验）。首先，我们把理想交流线和实际交流线中第一个任务所对应的数据点重叠起来。然后，我们对两个阶段的理想和实际交流线之间的差别做1000次数据采样，并计算出欧氏距离。每个组培训前后的两次实验结果都表现出了明显差异。Wilcoxon测试[①]表明，这些差别的统计显著值是0.05。对于这些对话的性质分析显示，在培训前后的谈话中参与者们都谈到了设计模式，但是在培训后的谈话中，关于设计模式的交流更加均衡一些。在培训前的实验中，参与者们虽然也使用模式术语，但是却对这些术语的意义知之甚少。

实验结果清楚地表明，各个小组的交流线在培训前后都起了变化。在对设计模式没有共同认识的情况下，解释阶段非常短甚至根本就没有，而随后的交流也常常会被某个组员单方面主导。但是，在三个月的设计模式培训后，我们可以明确地看出先是有解释阶段，然后才是交流均匀的工作阶段。即使是能力稍差一些的组员也能够同等地参与到设计的讨论中来。我们的实验结果支持了此前的假设，即设计模式能让团队成员之间的交流更加有效。很明显，把真正有用的设计概念放到记忆区块里确实可以改善交流的效率。

22.6　经验教训

设计和进行这一系列的实验也让我们学到了很多。设计这些实验的时候我们遇到了不少难

① Wilconxon测试评估的是两个样本是否是从同一个统计总体中取出的，统计总体无需是正态分布。如需更详细的介绍请参见参考文献[5]中的Mann-Whiteney测试。

题。我们曾经认为添加模式文档是非常简单的，但是要想“公正”地做出两套具备可比性的程序很难，因为我们设计的普通版程序总是和设计模式版的程序非常相似——我们这些研究人员的思想早已被“设计模式化”了！直到我们都确认普通版的程序比设计模式版的更简单但灵活性要稍微差一些（这正是我们想要的）之后，实验才得以继续。为了寻找记录交流的方法，我们也是绞尽脑汁。在阅读了大量实验设计的资料之后，我们决定使用对话分析法。但是又应该如何处理这些对话呢？我们最后采取了Barbara Unger的天才创意（见致谢部分），即使用理想交流线，并用它和实际交流线做对比。准备、执行和分析实验所需的时间大大超乎了我们的预料（花了差不多写三篇博士论文的时间），但是我们选择的时间点是合适的，当时设计模式这个话题还很新，参与者也很容易就找到了。在最后，实验的结果鼓舞人心。

Christensen那本关于实验设计的书[2]简直是老天的恩赐。这本书已经出到第10版了，内容涵盖了所有主要的实验方法。

当参与者需要进行多项任务的时候，我们推荐使用制衡性实验设计的方法。除了需要组织工作之外，制衡性实验设计并不费什么事，但是可以让我们检查是否存在学习效应的问题，这使我们更安心。此外，制衡实验设计还降低了抄袭的可能性，因为同桌的参与者来自不同的小组。（抄袭可能造成数据重复，使我们收集不到足够的数据，并有可能毁掉整个实验。）对于结果的分析，我们推荐使用免费的统计工具R。你可以在www.r-project.org下载。

下面我们的做法将和一般的有所不同。在统计检验中有“统计效力”的概念。统计效力指找到不同实验方法之间的区别（如果有的话）的概率。将这个概率保持在高位（比如80%）非常重要，否则的话，实验结果就有可能是无效的（这些实验得不出结果，因为得出结果的概率太低）。这时候统计效力分析就起作用了。简单来说分析的流程就是：在正式实验前先让少量参与者做一个测试，以此确定大概的结果范围。有这个信息，我们就能确定需要多少参与者才能达到预期的统计效力以及显著性水平。我们很幸运，预估的结果范围就足够大了，所以我们仅用现有的参与者就能够达到想要的结果。如果结果范围太小，那么需要的参与者数量很容易就飙升到80或者120了，而要想找到如此多的参与者非常困难。统计效力分析的方法在很多统计学的书籍中都有介绍，如参考文献[5]的第15章。很多统计工具（比如R）都提供了统计效力分析的工具。一定要在开始实验之前确定参与者的数量是否足够，否则最后你可能实验做不成，反而会一直停留在寻找更多参与者的阶段！

还有一个问题我们早前没有意料到，答案的正确性和参与者消耗的时间不能相互独立地进行分析。很明显，敷衍一个答案几乎不用花时间，但是好的答案则需要多一点时间。我们现在已经意识到盲目地单独比较工作时间是不合适的，反之必须在答案的质量一致的情况下再来对比消耗的时间。比如，可以设定三个正确度等级，然后按级别来比较消耗的时间。但是这有一个问题，就是所需的参与者数量。每个正确度的等级都需要足够数目的参与者来确保统计的显著性及效力，也就是说，我们将需要三倍的参与者。另一种方法是用验收检测的方式来确保正确度。参与者们不断地进行任务，直到提交的答案通过验收检测。如此，就能保证所有参与者的答案都满足最低正确度的要求。我们可以用自动化测试套件的方式来进行验收检测。参与者可以自行检测，也可以交由研究人员检测。虽然研究人员也可以在实验过程中亲自检查答案并在有不可接受的错

误的时候驳回，但是这样的做法会让研究的客观性受到一些影响，所以我们不推荐。一旦验收测试通过之后，研究人员就可以开始计算时间和其他需要的数据了。在验收测试之后，研究人员还可以使用一套更完善的方案来更详细地区分质量。关于最早使用验收测试的实验的详细情况，请参见参考文献[7]。

22.7　总结

编程是一件复杂的事情，所以单独的一个实验很难证明什么，除非是用来推翻一个假说。但是如果想要确定假说的正确性，我们就需要做更多的实验。我们用了三个实验（有两个经过了重复实验）来收集设计模式有效性的相关证据。第一组实验的结果表明，仅仅只是为设计模式加上文档，就可以改善程序员的效率并降低执行维护任务时的错误。在另一组对专业人士进行的实验中，我们将设计模式的解决方案和简单的临时性解决方案做了对比。在这些实验中，使用设计模式需要一定的额外工作，但是在大多数情况下这种额外工作被抵消了，因为维护变得更简单了。最后一个实验的结果显示，在对设计模式有共同认识的情况下，组员之间的交流得到了改善。在第一和第三个实验中，还有一些证据表明，在使用设计模式的情况下，能力稍差一些的程序员也能赶上经验更丰富的程序员。

这些结果有力地证明了设计模式能在维护工作中有效提升软件质量及程序员的效率。看来相信设计模式并没有错！

但是，设计模式对于软件的初始开发有没有积极的作用仍然有待考察。关于这个课题有一个相关的实验，这个实验从API的易用性的角度来分析了抽象工厂模式[3]。实验的结果表明，程序员在使用抽象工厂生成对象时消耗的时间比起直接用构造器消耗得多很多。但是，这次实验并没有发挥抽象工厂模式的长处：它测试的是构建单个对象，而不是处理一系列类似的对象。在实验中，这个模式无法发挥其（本应有的）优势。这个例子说明了要想设计行之有效的实验是多么困难。不过，我们还是需要更多的实验来测试设计模式在初始开发阶段的效果，而API易用性这个方向肯定是好的。

本章的三个实验都没有涉及使用设计模式可能节约的成本。这个问题就留待将来的实验解决吧。

22.8　致谢

我们的这些实验得到了很多人的帮助。首先，我最需要感谢的是匿名的参与者们。他们心甘情愿地忍受了实验中的奇怪程序和奇怪任务。感谢我早前的博士生，Lutz Prechelt、Michael Philippsen和Barbara Unger设计和测试了试验用的程序和维护任务，并收集和分析了相关的数据。感谢Ernst Denert和Peter Brossler鼓励sd&m的工程师们参加实验。感谢Lawrence Votta在实验设计上的帮助。感谢来自Simula实验室的Marek Vokac、Dag Sjoberg、Erik Arisholm以及Magne Aldrin提供了可以监测参与者的编程环境。在Simula实验室进行的实验的分析工作大部分由Maret Vokac进行。Dag Sjoberg提出了雇佣顾问来参与实验的（当时看来是疯狂的）想法，他还自己筹集了相关的资金。

22.9 参考文献

[1] [Chalmers 1999] Chalmers, Alan F. 1999. *What Is This Thing Called Science?*, Third Edition. Indianapolis, IN: Hackett Publishing Co.

[2] [Christensen 2007] Christensen, Larry B. 2007. *Experimental Methodology*. Boston: Allyn and Bacon.

[3] [Ellis et al. 2007] Ellis, Brian, Jeffrey Stylos, and Brad Myers. 2007. The Factory Pattern in API Design: A Usability Evaluation. *Proceedings of the 29th International Conference on Software Engineering*: 302-311.

[4] [Gamma et al. 1995] Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

[5] [Howell 1999] Howell, David C. 1999. *Fundamental Statistics for the Behavioral Sciences*. Pacific Grove, CA: Brooks/Cole Publishing Company.

[6] [Miller 1956] Miller, George A. 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review* 63: 81-97.

[7] [Müller 2004] Müller, Matthias. 2004. Are Reviews an Alternative to Pair Programming? *Empirical Software Engineering* 9(4): 335-351.

[8] [Parnas 1972] Parnas, David L. 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. of the ACM* 15(12): 1053-1058.

[9] [Prechelt et al. 2001] Prechelt, Lutz, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta. 2001. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering* 27(12): 1134-1144.

[10] [Prechelt et al. 2002] Prechelt, Lutz, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. on Software Engineering* 28(6): 595-606.

[11] [Unger et al. 2000] Unger, Barbara, Walter F. Tichy. 2000. Do Design Patterns Improve Communication? An Experiment with Pair Design. Workshop on Empirical Studies of Software Maintenance. *http://www.ipd.uni-karlsruhe.de/Tichy/publications.php?id=149*.

[12] [Vokac et al. 2004] Vokac, Marek, Walter Tichy, Dag Sjoberg, Erik Arisholm, Magne Aldrin. 2004. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment. *Empirical Software Engineering* 9: 149-195.

第23章

循证故障预测

Nachiappan Nagappan
Thomas Ball

实证软件工程（SE）的研究从软件工件及其相关的流程和变量中收集并分析数据，然后通过量化、定性等方式来探索不同变量与项目成功之间的关系。（这里的项目成功指在预算内按时交付高质量的安全软件。）在这一章中，我们会讨论Windows操作系统家族中故障预测的实证研究。 Windows是一个大型商业软件系统，主要用C / C ++ / C#开发，在全球有着数亿用户。有成百上千位工程师参与了这个项目，代码量超过四千万行。

故障预测是实证软件工程的一个重要部分，人们用它来了解软件维护所需的测试工作量和将来的资源分配，如下所述。

- 资源分配

  在每个大型软件开发中，软件质量保证（quality assurance）都会占用大量的时间。为了提高这项工作的有效性，需要提前为容易出错的模块制定修复计划。越容易出错的地方，越需要质量保证。

- 决策

  对故障数量的预测也能辅助其他的决定，比如选择正确的需求或设计、是否能选出最优的问题解决方法，等等。修改所伴随的风险（或修改之后出错的可能性）能帮助对修改所带来的风险做出决策。故障预测也可以用来评估系统的总体稳定性，帮助判断是否能按时完成项目。

我们会逐步讨论各种故障预测的指标。对于每一组指标，我们会讨论选择此指标的原因，对这一指标的相关描述，以及在大型商用项目上使用这些指标进行故障预测的结果。

23.1 简介

对于软件企业来说，越早估计产品质量越有利。因为我们只有在开发的中后期才能看出实际产品质量，那时修改缺陷的代价会很高[6]。IEEE软件工程术语的标准汇编[16]把瀑布式软件开发周期定义为“一个包含诸多活动的软件开发流程模型，通常有概念阶段、需求阶段、设计阶段、实现阶段、测试阶段、安装检验阶段，这些阶段按照顺序进行，也许会有重叠，但几乎没有迭代”。

我们可以在开发周期收集与质量相关的各种指标。我们的目标是利用这些指标，在开发周期的早期——开发和测试阶段，就能预测出产品发布之后可能发生的故障。 预测结果可以用来辅助重点测试、代码和设计的审查，并能以适当的代价指导缺陷的修正。

指标的选择基于实证技术，比如GQM原则[1]，以此来客观评估指标选择对于分析的重要性。内部指标（度量），如圈复杂度[19]，来自于产品本身。外部指标来自于对系统行为的外部评估。例如，现场发现缺陷的数量是一个外部指标。ISO / IEC标准[17]指出，一个内部指标只有在能证明与可见的外部属性相关时才有意义。当内部指标与现场的质量和可靠性有稳定的统计显著关系时，我们才可以把它作为早期指标来预测可见的外部产品质量。所以验证内部指标需要有令人信服的证据来证明：（1）该指标度量了希望度量的东西；（2）它与重要的外部指标相关联，如现场的可靠性、可维护性、易出错性[12]。

在这一章中我们会讨论六种不同的故障预测指标集，描述它们的内容，表明它们对于故障预测的重要性。我们也会提供一个微软商业案例研究的公开结果，以及相关的参考文献。最后，我们会对故障预测做出总结，并讨论一些未来的重要领域。六组预测缺陷的内部指标包括：

(1) 代码覆盖率；

(2) 代码变动；

(3) 代码复杂度；

(4) 代码依赖；

(5) 人员以及组织指标；

(6) 综合方法。

在Windows的案例中，我们评估的对象是二进制可执行文件。二进制文件是由源文件编译而来的exe、dll或sys文件。我们选择二进制文件的原因是：（1）现场故障能最直接地映射回二进制文件；（2）故障修复往往涉及几个文件的修改，大部分会被编译在一个二进制文件中；（3）二进制文件最起码能保证代码所有权，从而使结果更具有可操作性。从历史数据上看，微软用二进制文件作为度量单元，是因为它可以准确地把客户缺陷映射回来。

我们会从一个来自于微软Windows Vista或Windows Server 2003的故障预测案例分析中，为所有六组指标提供证据。我们会提供每一个与预测相关的查准率（precision）和查全率（recall），或者是准确率（accuracy）。查准率测量的是漏报率（false-negative），指易出错的二进制文件被归于不易出错的比率。查全率测量的是误报率（false-positive），指不易出错的二进制文件被归于是易出错的比率。这里我们专注的是故障预测的指标，而不是统计或机器学习①之类的技术指标，后者可以使用如逻辑回归、决策树、支持向量机等标准技术[14]。

如同所有的软件工程实验证据研究一样，要从软件工程的实证研究中得出通用的结论是很困难的，因为任何进展都很大程度上取决于有关的环境变量[3]。出于这个原因，我们不能预先假设在特定环境中得出的研究结果能推广到更普遍的情况[3]。如果一种理论能在不同的情况下得出相同的结果，那么研究者会对它更有信心[3]。所以我们鼓励读者在可能的情况下自己进行调查，在

① 机器学习：是人工智能的一个分支，关注于设计和开发一些算法，让计算机可以根据实证数据（如数据库中）自动“学习”行为。更具体地说，机器学习是一种用于创建数据集分析程序的方法。——译者注

他们的环境中重复这项研究之后再做出结论。我们希望本章可以用作一份文档，包含已经被证明在微软能够成功（或不成功）地预测故障的指标和功能集。

23.2　代码覆盖率

代码覆盖率是一项重要指标，可以用来量化测试范围。其背后的主要假设是，如果一个分支或一个语句包含一个缺陷，至少需要一个测试执行该语句或分支（并且所测结果与预期不符）才能发现缺陷。可以讨论的是，代码覆盖率越高，检测到的代码缺陷就越多，如果这些缺陷被修复了，那么产品的质量就会越好。该假设虽然被广泛认同，但是没有证据显示代码测试覆盖率越高，最终的故障会越少。

代码覆盖率越高质量越好这个论点有许多潜在的破绽。首先，覆盖率反映了被覆盖语句的百分比，但没有考虑这些语句是否有可能包含缺陷。因此，可能存在两个具有相同的覆盖率的测试集合，但两者检测到发布后缺陷的能力相差很多。其次，一个语句或分支被运行过并不意味着所有可能的数据都被运行过。用户的使用情况可能与测试人员设想的完全不同。用几个测试用例就可以覆盖的、控制流程简单的系统需要特别注意这个问题。在我们对代码覆盖率的研究中，基于与工程师的定性访谈（相对于定量），我们发现了造成此现象的几个相关因素[20]。

- 测试覆盖的代码不是正确的代码。
- 开发人员知道某些二进制文件是很复杂的，所以会给它们加上很高的测试覆盖率。但是，这些二进制文件也许是在系统中最经常被使用到的，因此也会发现更多的缺陷。也就是说，代码覆盖和使用模式不匹配。
- 考虑代码覆盖率时也应该考虑代码复杂度。例如，在圈复杂度为1000的二进制文件中实现80%的代码覆盖率，比在圈复杂度为10的二进制文件上实现80%的覆盖率困难。结合复杂度讨论代码覆盖率有助于充分使用该指标的功效。

尽管存在这些破绽，由于其在实践中的广泛使用，覆盖率似乎仍然是一个有前景的指标。然而，很少有出版物证明代码覆盖率与产品质量间的关系。 目前大多数这方面的研究存在内部有效性问题（比如对100行代码程序的研究）。为了找到代码覆盖率和质量关系间的证据，我们把Windows Vista（4千多万行代码、几千个二进制文件、数千名工程师）的分支覆盖率（branch coverage）与块覆盖率（block coverage）和其发布后六个月内的现场缺陷对应起来。我们观察到覆盖率和质量之间有弱的正相关性，预测查准率和查全率较差（查准率83.8%，查全率 54.8%；进一步阅读，请参见参考文献[20]）。对这个直接而简单的假设的调查结果使我们相信，代码覆盖率最好不要单独使用，而是需要与其他指标，如代码变动率，复杂度等一并考虑。

23.3　代码变动

随着时间的推移，需求的改变、代码优化、对安全性与可靠性问题的修复等原因，软件系统一直在演变。代码变动指标度量了一段时间内一个模块的改动，并量化了改动的程度。版本控制系统会自动记录下修改历史，我们很容易就可以从中得到相关的数据。大多数版本控制系统使用

文件比较工具（如Diff），自动估计一个程序员从旧版本到新版本的文件中添加、删除、修改了多少行代码。这些差异就是得出代码变动指标的基础。

要得到相对代码变动指标，就要对各种代码变动指标进行标准化。标准化参数包括代码的总行数、文件变动数量、文件总数量等。对于一个不断变化的系统，推荐使用相对方法来量化系统的变化。据我们研究所示，可以将这些相对指标设计成相互交叉检查，那它们就不会提供相互矛盾的信息。我们的基本假设是，在产品发布之前的同一段时间内，修改次数多的代码比修改次数少的代码更容易在发布之后发现缺陷。

我们分析了Windows Server 2003（W2k3）发布之后到W2k3服务包1（W2k3 - SP1）发布这段时间内的代码变动指标来预测W2k3-Sp1的缺陷密度。我们使用了直接收集到的代码变动指标，包括添加、修改和删除的代码行数（LOC），也收集了时间和文件数量来计算出一组相对代码变动指标[21]。

- 指标1：变动代码行数/总代码行数

  我们预计对于新的二进制文件来说，变动（增加+更改）代码的比率越高， 它的缺陷密度就越大。

- 指标2：删除代码行数/总代码行数

  我们预计对于新的二进制文件来说，删除代码的比率越高， 它的缺陷密度就越大。

- 指标3：变动文件数/文件总数

  我们预计二进制文件所包含的文件变动的比率越高，引入缺陷的可能就越大。例如，假设A和B两个二进制文件各包含20个文件。如果A有5个文件变动过，B有2个，我们认为A的缺陷密度会比B高。

- 指标4：代码变动次数/变动文件数

  假设A和B两个二进制文件，各包含20个文件，其中的5个变动过。如果A的5个文件变动过20次，而B的五个文件是变动过10次， 那么我们预测A会有更高的缺陷密度。指数4 也被用来交叉检查指数3的值。

- 指标5：代码变动所耗周数/文件总数

  指数5用于计算变动的时间范围。指标5如果较高，意味着花了较长的时间来修改数量较少的文件。这可能表明二进制文件可能包含复杂的文件，很难修改正确。因此，我们预计指数5越高，相关的二进制文件的缺陷密度也越高。

- 指标6：代码改变总行数/代码变动所耗周数

  代码改变总行数是指变动与删除的代码行数的总和。这个指标度量了代码随时间变化的程度，用以交叉检查指标5。代码变动所耗的周数并不能说明代码变动数量。指标6对应于我们的一种预期，即代码改变总行数越多，所耗的周数也会越长。指标6交叉检查指数5的值越高，缺陷密度也会越高。

- 指标7：变动代码行数/删除代码行数

  并不是所有的代码变动都是因为修复缺陷，指标7量化了新进行的开发。在功能开发时，变动的行数远远大于删除的行数，所以指标7较高则表示功能开发正在进行中。指标7可以交

叉检查指标1和指标2，这两者都不能准确地预测新功能的开发。

- 指标8：代码改变总行数/代码变动次数

  我们预计每次代码改变涉及的行数越多，缺陷密度就会越高。指标8可以交叉检查指标3和指标4，以及指标5和指标6。相对指标3 和指标4，指标8记录了实际发生的代码变动量。指标8交叉检查了文件是否因为小的修改而不停地变动。指标8还交叉检查了指标5和指标6，因为指标8的值越高（代码改变所涉及行数越多），每周工作的时间（指标5）和每周工作的代码行数会越高（指标6）。如果不是这样，那么大量的修改可能是在很短的时间内进行的，这就可能导致缺陷密度的提高。

在使用随机分组（random-splitting）预测W2K3 - SP1中容易出错和不易出错的二进制文件时，相对代码变动指标组成的预测模型达到了89%准确率。此外，对W2k3 - SP1缺陷密度的预测准确度也很高（如图23-1所示）。（为了保护机密数据，我们对数轴上的数据进行了标准化。）实线以上的部分表示高估值，以下的部分表示低估值。缺陷密度的估计值与实际值之间有明显的正相关性，并且显示出统计显著性，表明实际缺陷密度升高时，估计缺陷密度也在升高。

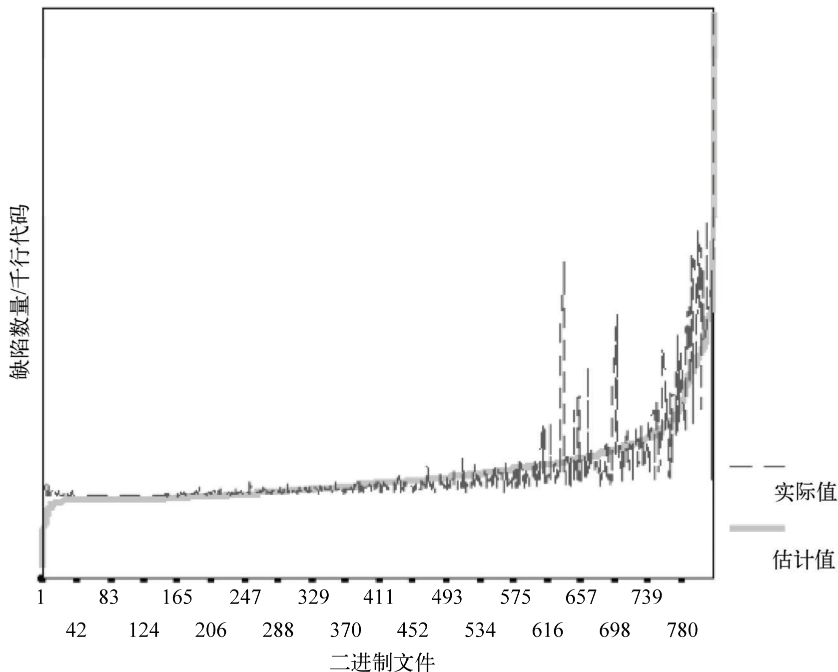


图23-1　系统缺陷密度的实际值与估计值[21]

这些结果加之先前公布的研究报告结果（例如，参考文献[21]和参考文献[26]中有一个更详细的清单）表明，代码变动和代码质量高度正相关。也就是说，在一个系统中，如果代码变动率较高则表明缺陷也更多。考虑到代码变动是新开发软件的重要组成部分（例如，产品的新功能），

我们也引入了相对代码变动指标，通过交叉检查来减轻这个问题。然而，我们的结果也清晰地表明，代码变动率必须与其他内部指标一起使用。

23.4 代码复杂度

讨论了代码变动之后，让我们分析一下代码复杂度。也就是说，一段代码的复杂度对预测缺陷能力的影响。复杂度可以用几个指标来量化，分为面向对象和非面向对象的指标，范围从传统的指标，诸如扇入扇出，到最新的CK指标[10]。CK指标组包括6个指标（主要被设计为面向对象的设计指标）：每类加权方法数（Weighted Methods per Class, WMC），对象之间的耦合（CBO），继承深度（DIT），子类数量（NOC），类的响应（RFC），方法间的低内聚度（LCOM）。基于先前发表的一些指标[10]，微软使用的面向对象和非面向对象的复杂度指标包括如下几项。

- 代码行数

  可执行的非注释代码行数。
- 圈复杂度

  圈复杂度[19]度量了一个程序单元内线性独立路径的数目。
- 扇入

  扇入指模块内某一个函数被其他函数调用的次数。
- 扇出

  扇出指模块内某个函数调用其他函数的次数。
- 函数数量

  一个类中的函数数量，包括公有（public）函数，私有（private）函数，受保护（protected）函数。
- 继承深度

  继承深度指一个类的最大继承深度。
- 耦合

  与其他类的耦合可以通过（a）类成员变量，（b）函数参数，（c）类成员函数体中定义的局部类，还有（d）基类和（e）返回类型。
- 子类数

  模块内直接继承自同一父类的子类数目。

在Windows Vista中，使用这些指标来预测缺陷的查准率为79.3%，查全率为66%。这些数据表明，虽然复杂度可以较好地预测代码质量，但不如代码变动指标。Khoshgoftaar等人做了一些相关的研究工作。他们研究了一个大型电信遗留系统（包含3.8万个函数，有171个模块）的连续两次发布，根据16个静态软件产品指标识别所有容易出错的模块[18]。当该模型被使用于第二次发布时，整体误判率为21.0%。El Emam等在大型的电信项目上研究了类的大小与缺陷的关系[13]。他们发现类的大小干扰了故障预测。CK指标也是以缺陷发生的背景为前提的。Basili等人用了8个学生项目来研究软件系统的出错倾向[2]。他们指出，WMC、CBO、DIT、NOC和RFC与缺陷相关，而LCOM与缺陷无关。Briand等人研究了一个业界案例，发现CBO、RFC、LCOM与类的易

出错性有关[8]。Tang等人研究了三个实时系统在测试及维护时发现的缺陷[28]。WMC和RFC的值越高，出错倾向也越高。进一步阅读，参见参考文献[23]、[8]、[4]。

23.5　代码依赖

至此我们已经知道代码变动与复杂度可以很好地预测缺陷。这也启发我们结合这两部分的信息来有效量化不同代码之间的关系。一般来说，在大型软件的开发中，一个优秀的软件架构可以让每个团队在不同的架构模块上独立工作。软件依赖指的是两部分代码之间的关联，比如数据依赖（模块A使用了模块B定义的一个变量）或是调用依赖（模块A调用了模块B定义的一个函数）。假设模块A与模块B之间依赖关系很多。如果模块B在不同版本之间变化很大，那么很有可能也需要修改模块A来适应模块B的修改。这意味着代码变动会随着代码依赖而传递。所以，高度代码依赖与代码变动会使错误波及整个系统，从而降低其可靠度。

为了研究用代码依赖预测缺陷的有效性，我们研究了Windows Vista二进制文件之间的依赖关系。我们统计了函数级的代码依赖，包括调用-被调用依赖、导入、导出、远程过程调用（RPC）、COM、注册表访问等。系统级依赖关系图可以被认为是一张Windows Vista的底层构架图。二进制文件之间的依赖被分为出、入两种（incoming dependency、outgoing dependency）。进一步的分类包括了二进制文件间的依赖总数，以及跨越多个模块的二进制文件数。用这样的依赖指标构建的模型所预测查准率与查全率分别为74.4%与69.9%。这与代码复杂度得出的结果相似，但没有代码变动得到的结果准确。关于Windows中用代码依赖与变化来预测缺陷的案例可以参照参考文献[22]。

在另一个相关研究中，Schröter等人发现导入依赖可以预测缺陷[27]，并提出了另一种方法来预测Java类中的缺陷。他们只着眼于一个类所使用的模块，而不是类本身的复杂度。在对Eclipse（开源集成开发环境）的研究中，他们发现使用编译包所造成的易出错率（71%）比使用GUI包（14%）大很多。Von Mayrhauser等人用一种自下而上的方法建立了一个缺陷构架模型，来研究软件构架的恶化与缺陷之间的关系[29]。该模型的构建结合了模块间的缺陷耦合度（degree of fault coupling）以及一个缺陷修复时涉及这两个模块的频率。研究结果指出了每个版本中最容易出错的关联，并指出同样的模块关联会重复出错，以此发现潜在的构架问题。

23.6　人与组织度量

康威法则声称：“设计系统的组织只会产出其自身沟通结构的翻版”[11]。Fred Brooks在《人月神话》一书中也提到产品质量受组织结构影响很大[9]。软件工程是一个复杂的工程活动。开发一个完整的产品牵涉到人、流程以及工具。实践中，商用软件开发一般由一些团队共同完成，每个团队的大小从几十人到上千人不等。通常，这些员工向组织结构中的一位经理或者几位经理汇报。我们一般通过度量代码变动、代码复杂度、代码覆盖率、代码依赖等来预测缺陷，但这样的方式忽视了软件开发中的另一个关键因素，那就是“人与组织结构”。

软件开发的全球化使得许多团队分散在世界各地，康威法则中所阐述的组织构架影响也日益彰显出来。本节中，我们使用八个指标来量化组织复杂度，以此分析组织构架与软件质量的

关系[25]。组织指标所度量的内容包括：开发人员之间的组织距离，工作于同一模块的员工数量，组织中多任务开发人员的数量，组织中某一个模块的修改次数。我们使用这些指标来预测Windows Vista中容易出错的二进制文件。部分指标如下所示（详细列表请参照参考文献[25]）。

- 工程师数量（NOE）

  接触过某个二进制文件的在职工程师的绝对数量。

- 前工程师数量（NOEE）

  接触过某个二进制文件，并且在软件发布之前就离职的工程师的绝对数量。

- 编辑频率（EF）

  二进制文件源代码的编辑总数。一次编辑指工程师从版本控制系统中签出代码、修改代码、然后签入回去。与每次编辑中修改代码的行数无关。

- 主负责人深度（DMO）

  该指标基于二进制文件编辑次数来决定其所有权等级。如果一个人的直属工程师执行了所有编辑的X%以上，那他在组织中的级别就是DMO。DMO指标基于二进制文件上的活动来决定其负责人。基于Windows的历史信息，我们把X%设为75%来量化负责度。

- 组织参与开发率（PO）

  汇报给DMO级别负责人的人数与DMO组织总人数的比率。

- 组织级代码负责等级（OCO）

  二进制文件负责人所处的组织对二进制文件编辑的百分比，如果没有负责人的话，就指编辑最多的组织。

- 总体组织负责度（OOW）

  修改某二进制文件的人员中，处在DMO级别的人数与所有参与修改人数的比率。该指数越高越好。

- 组织交叉因子（OIF）

  参与10%以上修改的组织数量，度量级别是总体组织负责人。

提出这些指标是为了平衡组织结构对于质量影响的各种假设，有些假设似乎是相互对立的。这些假设的高度概括的说明与其对应的量化指标在表23-1中显示。

表23-1 组织指标概要[25]

假 设	指 标
一份代码上工作的人越多，质量越差	NOE
团队成员流失越高，知识流失率也越高，因此会影响质量	NOEE
模块修改得越多，稳定性越差，质量越低。	EF
负责人层次越低，质量越好	DMO
员工越集中（组织结构上），质量越高	PO
修改代码的人员越集中，质量越高	OCO
接触代码的人员越分散，质量越低	OOW
接触代码的组织越多，质量越低	OIF

我们用Vista代码做了50次随机切分，得到了图23-2中的数据。该图显示了预测查准率与查全率的一致性和连贯性。数据的波动很小，表示预测模型的有效性，即组织指标能够预测软件质量。查准率与查全率分别为86.2%与84%。与之前的研究相比，这是我们得到的最高查准率与查全率。这就可以证明团队的组织构架是预测和理解软件质量的一个重要因素。因为组织构架指数比代码指标更能反应出代码质量。同时也意味着建立适合的组织与团队构架十分关键。这些结果还说明了组织构架对于质量的影响以及重组团队的计划的重要性，以降低其对产品质量的负面影响。

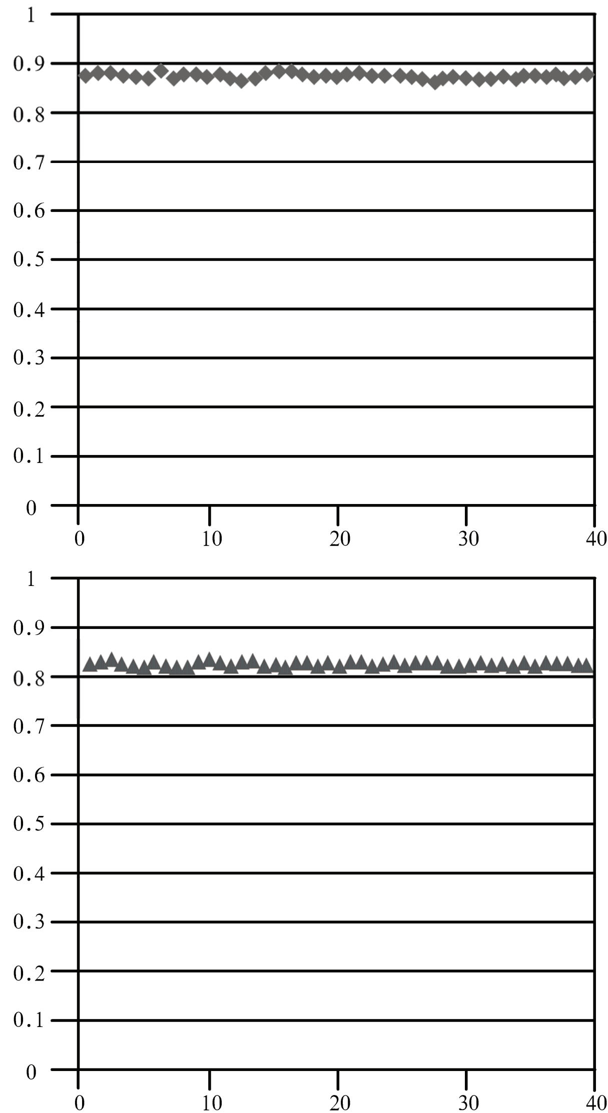


图23-2　组织构架预测的查准率与查全率[25]

23.7 预测缺陷的综合方法

目前为止，我们已经单独分析了一系列度量指标，包括复杂性度量、代码变动、代码覆盖率，等等。在本节中我们将讨论如何综合利用这些度量方法，使之成为更强的预测缺陷的工具。

图23-3展示了工程师们工作于不同的二进制文件的简单网络图。图中显示了不同网络之间的代码依赖关系。该图在一个网络中整合了人、变化率（修改/代码贡献）、代码依赖的信息。

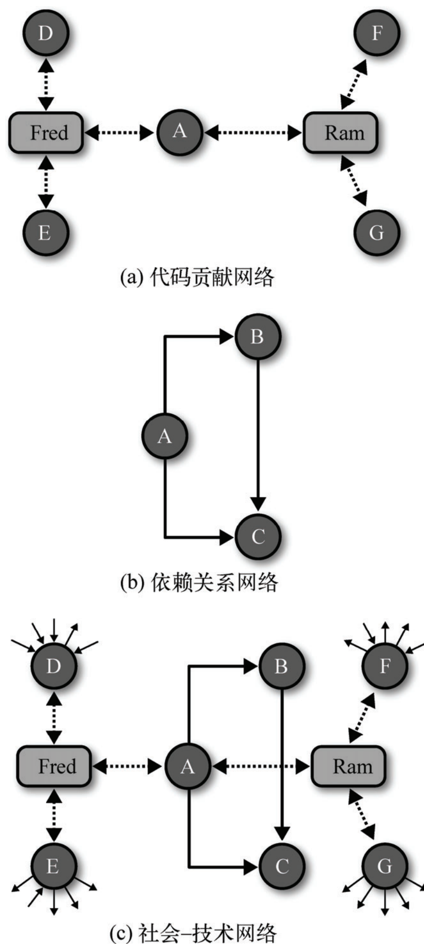


图23-3 社会–技术网络图[5]

我们设计了这样一个网络来集成Windows Vista中人、代码变动贡献和依赖关系的信息。之后会介绍几个社会网络的度量方法，用它们绘制Windows Vista的社会网络[5]（类似于图23-3）。

自我网络度量[7]基于某个节点的邻居。被评估的节点表示自我，它的邻居包括自我、与自我连接的节点以及这些节点之间的所有边。

- 规模（size）

  自我网络中的节点数。
- 连结（ties）

  网络中所有连接节点之间的边数。
- 配对（pairs）

  个体关系网络中可能存在的有向边的个数。
- 密度

  实际存在的边相对于可能存在的边的比率（连结/配对）。
- 弱模块

  弱关联模块的数量。
- 标准化弱模块

  用规模标准化的弱关联模块（弱关联模块数量/规模）。
- 两步区域

  距离自我（ego）不超过两跳的节点比率。
- 到达效率

  用网络规模标准化的两步区域（到达效率高意味着该自我的主要联系人在网络内影响很大）。
- 中间人

  仅用自我连结的节点配对数（这样，自我在配对间完全充当中间人的角色）。
- 标准化中间人

  用配对数标准化中间人。
- 自我的中介度（betweenness）

  个体在个体关系网络中的中介度。

我们在Vista网络上完整计算了这些社会网络指标的值（包括开发人员、代码贡献与依赖关系），并用这些指标作为输入建立预测模型。我们发现如果配合使用代码依赖网络的话，预测的查准率与查全率会更高。IBM Eclipse的相关实验也得出了相似的结果[5]。表23-2展示了查准率、查全率与模型拟合的F分值，也说明了社会–技术网络能更好地预测故障。表23-2中的“综合模型”给出了一种只结合“贡献”网络（团队合作）与“依赖网络”（依赖关系）的模型，与社会－技术网络形成对比。有关代码复杂度、代码变动、代码覆盖等指标联合预测故障的其他内容可以参照参考文献[24]。

表23-2 社会–技术网络模型在不同版本Eclipse上的总体效率

版 本	网 络	查 准 率	查 全 率	F-Score	Nagel.①
2.0	依赖关系网络	0.667	0.779	0.705	0.532
	代码贡献网络	0.808	0.854	0.824	0.702

① Nagelkerke决定系数（Nagelkerke’s coefficient of determination）：统计学中，用来度量统计模型多大程度上能预测未来结果的一个参数。——译者注

（续）

版 本	网 络	查 准 率	查 全 率	F-Score	Nagel.
	综合网络	0.826	0.814	0.813	0.909
	社会技术网络	0.755	0.859	0.800	0.747
2.1	依赖关系网络	0.693	0.753	0.710	0.626
	代码贡献网络	0.675	0.780	0.719	0.607
	综合网络	0.755	0.777	0.758	0.805
	社会技术网络	0.747	0.809	0.770	0.689
3.0	依赖关系网络	0.631	0.737	0.673	0.494
	代码贡献网络	0.681	0.683	0.673	0.353
	综合网络	0.745	0.756	0.743	0.616
	社会技术网络	0.767	0.777	0.769	0.600
3.1	依赖关系网络	0.579	0.718	0.634	0.391
	代码贡献网络	0.639	0.646	0.629	0.295
	综合网络	0.693	0.796	0.735	0.689
	社会技术网络	0.820	0.800	0.806	0.668
3.2	依赖关系网络	0.698	0.780	0.731	0.495
	代码贡献网络	0.614	0.720	0.654	0.371
	综合网络	0.835	0.866	0.846	0.816
	社会技术网络	0.793	0.784	0.785	0.572
3.3	依赖关系网络	0.693	0.743	0.711	0.433
	代码贡献网络	0.725	0.669	0.688	0.356
	综合网络	0.742	0.780	0.754	0.686
	社会技术网络	0.820	0.831	0.823	0.727

23.8 结论

在本章中我们基于Windows案例讨论了各种预测故障的方法。表23-3总结了Windows Vista中各种指标预测故障的查准率与查全率。

表23-3 不同软件指标模型总体准确率

模 型	查 准 率	查 全 率
组织结构	86.2%	84.0%
代码变动	78.6%	79.9%
代码复杂度	79.3%	66.0%
社会网络/综合指标	76.9%	70.5%
依赖关系	74.4%	69.9%
代码覆盖率	83.8%	54.4%

此外，综合几种指标得出的社会网络指标也能够正确地预测故障。正如之前所述，所有实证研究和证据都是在微软搜集的，很多来自于Windows项目。有这些结果并不适用于其他软件产品或者开发环境。只有在不同的环境中进行相似的测试之后，这里有关故障预测的证据才能被证明具有还是具有普遍适用性。请参阅补充文章“如何一步步建立质量预测器”[23]。本章节的主要目的是介绍我们所使用的各种预测指标，以及在微软个案中得到的结果和经验。我们希望以此来鼓励研究者在其他项目中重复这些方法，分析检验它们在不同项目中的表现。

如何一步步建立质量预测器

(1) 选择所研究的软件E。E可以是一个较早发布的产品，或者一个相似项目。

(2) 把E分解为可以度量质量（二进制文件、模块）的一个个实体（子系统，模块，文件，类）$E=\{e_1,e_2,e_3\cdots\}$

(3) 建立一个方法 $E\to R$ ，为E中的每一个实体设定一个质量值。这里需要参考版本与缺陷的历史数据。

(4) 建立一个度量方法的集合 $M=\{m_1,m_2,m_3\cdots\}$ ，其中每一个 $m\in M$ 是一个映射 $m:E\to R$ ，为每一个 $e\in R$ 设定一个度量值。M需要根据项目和编程语言所定。

(5) 对于每一个 $m\in M,e\in E$ ，定义 $m(e)$ 。

(6) 找出 $m(e)$ 与 $defects(e)$ 的关系，以及 $m(e)$ 相互之间的联系。

(7) 如果需要，可以使用主成分分析（principal component analysis），找出主成分集合 $PC=\{pc_1,pc_2,pc_3\cdots\}$ ，其中每一个 $pc_i\in PC$ ， $pc_i=\langle c_1,c_2,\cdots,c_{|M|}\rangle$ 。

(8) 使用主成分PC为 $E'=\{e_1',e_2',e_3'\cdots\}$ 建立一个预测器。同时评估其解释力与预测力。

本章中的指标都是自动采集的，读者们也可以用商业软件工具或者开源集成环境中的工具来自动得到相应的指标。图23-4①是Microsoft Visual Studio的截图，图23-5②是IBM Eclipse插件的截图。

Code Metrics Results

Filter: None　Min:　Max:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Library - Debug (Library.dll)	119.81	31.00	3.00	27.67	63.33
<PrivateImplementationDetails>{5E05BD25-B07F-42:	171.00	0.00	2.00	0.00	0.00
<Module>	171.00	0.00	1.00	0.00	0.00
Library	17.42	93.00	6.00	83.00	190.00
ProgramLibrary	17.42	93.00	6.00	83.00	190.00
DoEvenMoreWork(System.String,System.Int6	-35.83	223.00		82.00	604.00
DoSomeWork(System.String,System.Int64):S	35.28	57.00		20.00	137.00
.ctor()	170.77	1.00		1.00	0.00
SimpleProgram - Debug (TestAssembly1.dll)	143.17	1.00	1.50	2.50	3.50
<Module>	171.00	0.00	1.00	0.00	0.00
SimpleProgram	115.34	2.00	2.00	5.00	7.00
Program	115.34	2.00	2.00	5.00	7.00
Main(System.String[]):System.Void	101.33	3.00		4.00	15.00
VerifyArguments(System.String[]):System.Bo	105.60	4.00		2.00	12.00
.ctor()	170.77	1.00		1.00	0.00

图23-4　Microsoft Visual Studio™中的指标报告

① 出自http://blogs.msdn.com/b/codeanalysis/archive/2007/02/28/announcing-visual-studio-code-metrics.aspx。2010年7月13日获得。

② 出自http://metrics.sourceforge.net/。2010年7月13日获得。

Metrics - /net.sourceforge.metrics

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Packages	16					
Number of Methods (avg/max per type)	1310	6.65	8.553	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
tgsrc	489	7.191	11.544	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
src	761	6.238	6.553	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core.sources	108	15.429	12.129	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui	77	9.625	10.111	33	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core	198	6.6	7.093	27	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.preferences	52	6.5	7.467	26	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.dependencies	95	5.588	3.727	15	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.persistence	18	4.5	4.33	12	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler.implementa...	54	5.4	2.871	10	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.xml	41	4.1	2.022	9	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.calculators	79	4.158	2.254	8	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.propagators	31	5.167	1.067	7	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.tests	8	2.667	1.886	4	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler	0	0	0			
classycle	60	8.571	2.556	13	/net.sourceforge.metrics/classycle/classycle/g...	
Lines of Code (avg/max per type)	6593	33.467	49.02	339	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
Number of Interfaces (avg/max per packageFragment)	16	1	1.414	4	/net.sourceforge.metrics/src/net/sourceforge/...	
Lines of Code (avg/max per method)	6593	4.812	7.355	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
classycle	324	5.4	9.94	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
tgsrc	2321	4.661	8.278	59	/net.sourceforge.metrics/tgsrc/com/touchgrap...	scrollSelectPanel
src	3948	4.862	6.473	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
net.sourceforge.metrics.ui	544	6.8	8.707	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable.java	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
setMetrics	52					

图23-5 IBM Eclipse的指标插件

23.9 致谢

感谢在微软公司和Windows项目中的所有同事对本研究的帮助。要特别感谢我们所有的作者，Thomas Zimmermann、Brendan Murphy、Vic Basili、Andreas Zeller、Audris Mockus, Prem Devanbu和Chris Bird。

23.10 参考文献

[1] [Basili et al. 1994] Basili, V., G. Caldiera, and D.H. Rombach. 1994. The Goal Question Metric Paradigm. In *Encyclopedia of Software Engineering*, ed. J. Marciniak, 528-532. Hoboken, NJ: John Wiley & Sons, Inc.

[2] [Basili et al. 1996] Basili, V., L. Briand, and W. Melo. 1996. A Validation of Object Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering* 22(10): 751-761.

[3] [Basili et al. 1999] Basili, V., F. Shull, and F. Lanubile. 1999. Building Knowledge through Families of Experiments. *IEEE Transactions on Software Engineering* 25(4): 456-473.

[4] [Bhat and Nagappan 2006] Bhat, T., and N. Nagappan. 2006. Building Scalable Failureproneness Models Using Complexity Metrics for Large Scale Software Systems. *Proceedings of the XIII Asia Pacific Software Engineering Conference*: 361-366.

[5] [Bird et al. 2009] Bird, C., et al. 2009. Putting It All Together: Using Socio-technical Networks to Predict Failures. *Proceedings of the 20th IEEE international conference on software reliability engineering*: 109-119.

[6] [Boehm 1981] Boehm, B.W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

[7] [Borgatti et al. 2002] Borgatti, S., M.G. Everett, and L.C. Freeman. 2002. *UCINET 6 for Windows: Software for Social Network Analysis*. Harvard, MA: Analytic Technologies.

[8] [Briand et al. 1999] Briand, L.C., J. Wuest, S. Ikonomovski, and H. Lounis. 1999. Investigating quality factors in object-oriented designs: an industrial case study. *Proceedings of the 21st international conference on software engineering*: 345-354.

[9] [Brooks 1995] Brooks, F.P. 1995. *The Mythical Man-Month*, Anniversary Edition. Boston: Addison-Wesley.

[10] [Chidamber and Kemerer 1994] Chidamber, S.R., and C.F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20(6): 476-493.

[11] [Conway 1968] Conway, M.E. 1968. How Do Committees Invent? *Datamation* 14(4): 28-31.

[12] [El Emam 2000] El Emam, K. 2000. *A Methodology for Validating Software Product Metrics*. Ottawa, Ontario, Canada: National Research Council of Canada.

[13] [El Emam et al. 2001] El Emam, K., S. Benlarbi, N. Goel, and S.N. Rai. 2001. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering* 27(7): 630-650.

[14] [Han and Kamber 2006] Han, J., and M. Kamber. 2006. *Data Mining: Concepts and Techniques*, Second Edition. San Francisco: Elsevier.

[15] [Herbsleb and Grinter 1999] Herbsleb, J.D., and R.E. Grinter. 1999. Architectures, coordination, and distance: Conway's Law and beyond. *IEEE Software* 16(5): 63-70.

[16] [IEEE 1990] IEEE. 1990. *IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*. New York: The Institute of Electrical and Electronics Engineers.

[17] [ISO/IEC 1996] ISO/IEC. 1996. *DIS 14598-1 Information Technology—Software Product Evaluation.* International Organization for Standardization.

[18] [Khoshgoftaar et al. 1996] Khoshgoftaar, T.M., E.B. Allen, N. Goel, A. Nandi, and J. McMullan. 1996. Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System. *Proceedings of the Seventh International Symposium on Software Reliability Engineering*: 364.

[19] [McCabe 1976] McCabe, T.J. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* 2(4): 308-320.

[20] [Mockus et al. 2009] Mockus, A., N. Nagappan, and T.T. Dinh-Trong. 2009. Test Coverage and Post-verification Defects: A Multiple Case Study. *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*: 291-301.

[21] [Nagappan and Ball 2005] Nagappan, N., and T. Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. *Proceedings of the 27th international conference on software engineering*: 284-292.

[22] [Nagappan and Ball 2007] Nagappan, N., and T. Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*: 364-373.

[23] [Nagappan et al. 2006a] Nagappan, N., T. Ball, and A. Zeller. 2006. Mining metrics to predict component failures. *Proceedings of the 28th international conference on software engineering*: 452-461.

[24] [Nagappan et al. 2006b] Nagappan, N., T. Ball, and B. Murphy. 2006. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. *Proceedings of the 17th International Symposium on Software Reliability Engineering*: 62-74.

[25] [Nagappan et al. 2008] Nagappan, N., B. Murphy, and V. Basili. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. *Proceedings of the 30th international conference on software engineering*: 521-530.

[26] [Ostrand et al. 2004] Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2004. Where the Bugs Are. *Proceedings of the 2004 ACM SIGSOFT international symposium on software testing and analysis*: 86-96.

[27] [Schröter et al. 2006] Schröter, A., T. Zimmermann, and A. Zeller. 2006. Predicting Component Failures at Design Time. *Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering*: 18-27.

[28] [Tang et al. 1999] Tang, M.-H., M.-H. Kao, and M.-H. Chen. 1999. An empirical study on object-oriented metrics. *Proceedings of the 6th International Symposium on Software Metrics*: 242.

[29] [von Mayrhauser et al. 1999] von Mayrhauser, A., J. Wang, M.C. Ohlsson, and C. Wohlin. 1999. Deriving a fault architecture from defect history. *Proceedings of the 10th International Symposium on Software Reliability Engineering*: 295.

第24章

采集缺陷报告的艺术

Rahul Premraj
Thomas Zimmermann

孩子们喜欢虫子，有的孩子还把捉到的虫子作为战利品放在罐子里面收藏起来，那可是他们的珍宝。经过了一段时间，孩子们可以收集到大量各种各样不同种类的虫子。有些孩子会研究自己搜集的虫子，并且根据它们的特征，比如形状、大小、颜色、脚的数量、是否会飞，来标识它们。孩子们对虫子的珍视程度也因为它们各自的稀有程度和捕获的难度而有所不同。收藏可能存在重复，但是即使是重复的虫子也不会是一模一样的，因为外形和大小的差别会很大。

可是作为程序员的我们却不喜欢虫子（bug[①]）。我们希望软件里面没有缺陷，如果有的话，一旦找到就要把它们消灭掉！但不幸的是，消灭缺陷（更客气的说法，回应软件修改请求）并不是简单的事情。程序员们必须仔细研究缺陷相关的问题，针对如何解决缺陷做一个彻底的调查，检验它的副作用，最终决定如何采取行动并付诸于实施。这是一个困难的任务，就和世上的虫子一样，软件缺陷也各种各样。通常在项目缺陷数据库中的缺陷是被单独研究的，因为与其他缺陷相比，它们对整个软件系统的影响，以及它们的起因、位置、严重程度都不同。经过一段时间，项目中会发现重复的缺陷，就好像收集到的虫子中同一种类的会有好多个。最终，几乎每一个项目的缺陷都会多到来不及修复，正如虫子太多一个人抓不过来一样。

所以软件缺陷的研究是值得好好花一些时间的。而其中的第一步就是对缺陷报告中报告者提供信息的研究。本章中，我们将讨论优秀缺陷报告需要具备的要素，以及搜集缺陷报告的意义。

24.1　缺陷报告的优劣之分

当用户在软件系统中碰到一个缺陷时，他们会汇报给开发人员，希望尽快将这个缺陷修复。开发人员从缺陷报告中提供的信息来仔细了解某个问题，常常也受这些信息的启发找到缺陷的源头。

但是缺陷报告的信息质量和详细程度却各有不同。让我们看一下Eclipse项目中编号为#31021的缺陷报告。

① bug在软件工程里面意味着软件缺陷。——译者注

I20030205

执行下列代码。双击目录树上的一项，它没有展开。

注释掉选择监听器的代码，然后双击目录树上的一项，它才展开。

```
public static void main(String[] args){
  Display display = new Display();
  Shell shell = new Shell(display);
  [. . . ] (21 lines of code removed)
  display.dispose();
}
```

报告者提交了一份代码范例，仔细说明了如何运行这份代码来重现这个缺陷。一旦程序员重现并观察到了这个缺陷，研究其产生原因可能会方便很多。

在另一方面，下面一份缺陷报告也来自Eclipse项目（#175222），事实上它本身不是一个缺陷，却被错误地归为缺陷。

> 我想在Eclipse中用CDT创建一个新的插件。这当然是可能的。我已经在Eclipse文档生成了一个RD文件。我知道如何使用Java创建一个插件。但是，我想用CDT创建一个新的插件（用户定义插件），然后自己用程序实现。这可能吗？谁来帮助我一下。

缺陷报告中的信息质量可以严重影响一个问题的解决以及解决它所用的时间。缺陷报告如果含有所有的必要信息，可以在一定程度上降低修复该缺陷的难度。与此相反，缺陷报告中的信息不充分可能会延迟缺陷修复，因为开发人员需要自己找出缺失的信息或者联系提交者以便获得更多信息。

24.2 优秀缺陷报告需要具备的要素

通过网上问卷调查，我们询问了150名开发人员怎样的缺陷报告会帮助他们解决缺陷，他们分别来自三个最大也是最成功的开源项目（Apache、Eclipse以及Mozilla）。我们还询问了缺陷报告者在缺陷报告中提供了哪些信息，哪些信息是最难提供的。

调查结果如表24-1中所示。针对开发人员的调查包括以下两类问题。

- 缺陷报告的内容如下所示。
  - ❑（D1）开发人员使用了下列选项中的哪些来修复缺陷？
  - ❑（D2）哪三个信息对开发人员的帮助最大？

  这个问题的答案可以辅助人们开发工具或编写指南来帮助报告者在缺陷报告中提供开发人员需要的详细信息。我们为开发人员提供了16个选项，一些是来自Mozilla缺陷撰写指南①，还有一些来自Bugzilla数据库中找到的标准属性。对于第一个问题（D1），开发人员可以自由地选择，没有数量的限制。第二个问题（D2）最多选择三项，以体现他们选择的重要性。

① Mozilla缺陷撰写指南描述了有效缺陷报告的准则（比如准确、清晰、每个报告说明一个缺陷等），并且列出了对每个缺陷报告都尤为重要的一些信息，比如重现的步骤、实际结果、期望结果[6]。

- 缺陷报告的问题如下所示。
  - ❑（D3）开发人员在修复缺陷中遇到哪些问题？
  - ❑（D4）在修复缺陷中，大多数延迟是由哪三大因素引起的？

  我们问这些问题的动机是为了找到开发人员所面临的突出障碍，以便日后更加小心地对待这些问题，甚至实现自动化报告缺陷。典型问题如：缺陷报告中提供了不正确的信息，比如错误的操作系统。

  缺陷报告的其他问题包括语言使用问题（模糊），重复缺陷以及不完整的信息。最近垃圾缺陷函件也成了一个问题，特别在TRAC问题跟踪系统中。我们这里不讨论把缺陷指派给错误的开发人员，因为缺陷汇报者对缺陷分类的影响很小。总的来说，我们为开发人员提供了21个选项。对于第三个问题（D3），我们一再强调，开发人员的选择数目是不受限制的。但对于第四个问题，选择限于三个。表24-1包含了所有选项。

  我们问了缺陷报告者三个问题，分成以下的两类（请参阅表24-1）。

- 缺陷报告的内容。
  - ❑（R1）报告者在报告缺陷时提供了哪些信息?
  - ❑（R2）哪三个信息最难提供的？

  我们提供给报告者的16个选项与提供给开发人员的一致。这是为了检查报告者提供的信息是否与开发人员经常使用的或者他们认为重要的信息一致（通过比较D1、D2与R1的回答）。第二个问题帮助我们发觉哪些问题是很难收集的，哪些工具可以更好地用来帮助解决这些问题。在R1中，报告者的选择不受限制，但R2问题的选择限制在三个之内。

- 相关内容。
  - ❑（R3）哪三个信息是报告者认为对于开发人员来说最重要的信息？

  这里的选项也与提供给开发人员的一致，用来检查报告者与开发人员想法是否一致（比较D2与R3的答案）。对于这个问题（R3），报告者至多可以选择三个答案，但可以选择任何选项，无论他们是否已经在R1中选择了这些选项。

此外，对于开发人员与缺陷报告者，我们都询问了他们与缺陷报告相关的想法和经验（D5与R4）。

表24-1　给Apache、Eclipse和Mozilla开发人员（Dx）和缺陷报告者（Rx）的问卷

缺陷报告内容	D1：你使用了下列选项中哪些来修复缺陷 D2：哪三个选项对你的帮助最大 R1：你在报告缺陷时提供了哪些选项 R2：哪三个选项最难提供的 R3：你认为哪三个选项对开发人员修复缺陷最重要			
	❑ 产品信息	❑ 硬件信息	❑ 观察到的行为	❑ 截图
	❑ 组件信息	❑ 操作系统信息	❑ 期望行为	❑ 代码范例
	❑ 版本号	❑ 总结	❑ 重现步骤	❑ 错误报告
	❑ 严重程度	❑ build信息	❑ 栈跟踪信息	❑ 测试用例

（续）

缺陷报告的问题	D3：你在修复缺陷中遇到哪些问题			
	D4：大多数缺陷修复的延迟是由哪三大因素引起的			
	已有信息	存在错误的信息在于	报告者使用了	其他
	❑ 产品信息	❑ 代码范例	❑ 语法错误	❑ 重复错误
	❑ 部件信息	❑ 重现步骤	❑ 没有条理的描述	❑ 垃圾函件
	❑ 版本号	❑ 测试用例	❑ 随意的描述	❑ 不完整的信息
	❑ 操作系统信息	❑ 栈跟踪信息	❑ 描述过长	❑ 病毒、蠕虫
	❑ 观察到的行为		❑ 非技术语言	
	❑ 期望行为		❑ 拼写错误	
评论	D4/D5欢迎分享你关于缺陷报告的意见与经验			

24.3 调查结果

图24-1是开发人员的调查结果，图24-2是缺陷报告者的调查结果。

图中，每一项的调查结果用横条来表示（▬▬），其含义解释如下（用D1与D2作为例子）：

- ❑ 有颜色部分（▬+▬）表示该选项在D1中被选择的次数；
- ❑ 黑色的部分表示该选项在D1与D2问题都被选中的次数。

黑色比灰色的比率越大，表示该选项对于开发人员越重要。每一个选项的重要性在括号内表示，比如在图24-1中，开发人员认为重现缺陷的步骤（▬▬，83%）比build信息（▬▬，8%）更加重要。

24.3.1 开发人员眼中的缺陷报告内容

如图24-1所示，对于各类项目，最常使用的信息包括重现缺陷步骤，观察到的行为与预期行为，栈跟踪信息和测试用例。最不常被开发人员使用的项目包括硬件信息与严重程度。Eclipse与Mozilla的开发人员比较喜欢截图，而Apache与Eclipse的开发人员则更喜欢用代码范例与栈跟踪信息。

显而易见，重现缺陷的步骤是最重要的信息。其后是栈跟踪信息与测试用例，可以用它们缩小导致缺陷的代码范围。观察到的行为与重现缺陷步骤类似，也较为重要。截图虽然也被认为相当重要，但它一般只适用于界面错误等子类缺陷。

其他一些信息的重要性相对来说较低，比如期望行为、代码范例、总结以及一些必须填写的信息比如版本号、操作系统、产品和硬件信息。正如Mozilla开发人员所认为的那样，并不是所有的项目都需要用到这些信息：

> “因为我们项目的很多缺陷出现于各种平台，所以产品信息，甚至部件信息都是无用的，在一定程度上硬件与操作系统信息也是不需要的。”

无论如何，我们要谨慎地解释这里的结果。在调查中重要性低的信息并非完全无用，我们可能仍然需要它们来理解、重现、鉴别缺陷。

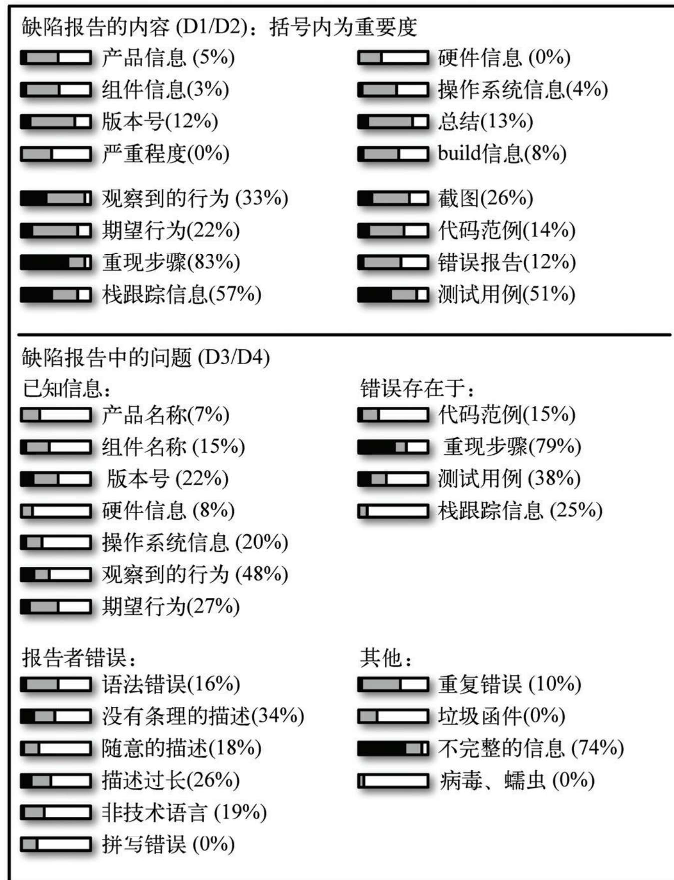


图24-1　开发人员调查结果（130份来自Apache、Eclipse与Mozilla开发人员的可靠调查反馈）

24.3.2　报告者眼中的缺陷报告内容

图24-2的第一部分是报告者提供的信息。和预测的一样，观察到的行为与预期行为以及重现缺陷的步骤是前三甲。只有少部分用户在缺陷报告中加入了栈跟踪信息，代码范例和测试用例，原因可能是因为人们很难提供这些信息，括号内的数字表示难度，是调查对象选择该项目难以提供的比率。栈跟踪信息，代码范例和测试用例是最难提供的三个信息，以测试用例为首。出人意料的是，重现缺陷的步骤与部件信息也较难提供。对于部件信息，报告者在注释中指出他们常常很难确认缺陷是在哪一部件中发生的。

在报告者认为对开发人员最有帮助的信息中，重现缺陷的步骤与测试用例排名最高。比较测试用例在三个问题中的结果后发现绝大多数报告者认为它是十分重要的，但只有少数报告者提供了该消息，因为它很难提供。这启发我们可以将记录捕获、重播测试用例的工具整合入缺陷跟踪

系统中。栈跟踪信息也有类似的情况，它们多数隐藏在日志中，很难找到。另一方面，开发人员与报告者都认为部件信息是次要的，而且很难提供。

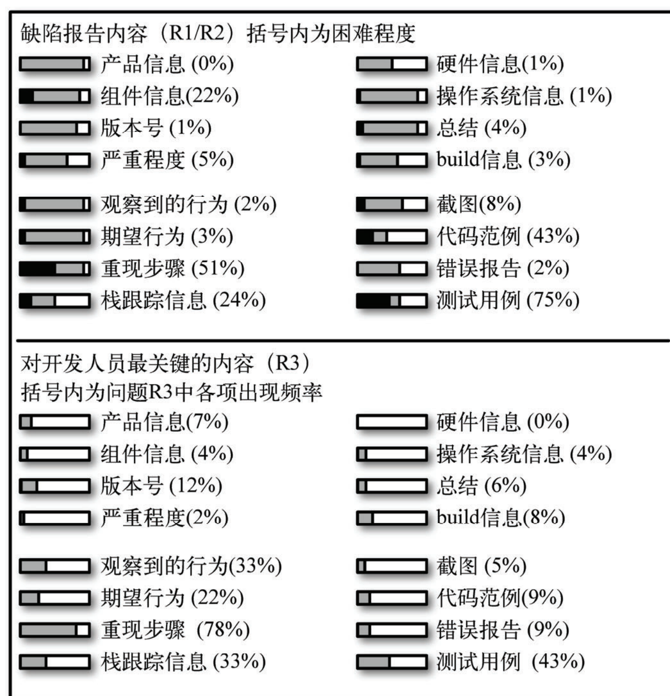


图24-2　报告者的调查结果（215份来自于Apache、Eclipse和Mozilla的报告者的可靠问卷反馈）

24.4　来自不一致信息的证据

我们比较了开发人员与报告者的调查结果，来查看两方面对于缺陷报告中的重要部分的看法是否一致。

首先，我们比较开发人员用来解决缺陷的信息（问题D1）与报告者提供的信息（R1）。在图24-3中，左列是降序排列的开发人员使用该信息的百分比，右列是降序排列的报告者提供该信息的百分比。两列中的相同项目用线连接起来，显示该项目在开发人员和报告者两方面的一致与否。图24-3中显示只有前三个信息以及最后一个信息一致。中间的都不一致，而其中栈跟踪信息、测试用例、代码范例、产品信息、操作系统信息最为显著。总体来说，开发人员使用信息与报告者提供信息之间的斯皮尔曼等级相关系数[①]（Spearman correlation）是0.321，远非理想值。

① 斯皮尔曼等级相关系数衡量两个变量之间关联强度，从−1到+1。如果值趋近于1或者−1，表明关联紧密，如果等于0，则表示没有关联。加减号表示是正向关联还是逆向关联。

然后，我们检查了报告者是否提供了开发人员认为最重要的信息。图24-4中，左列对应开发人员认为重要的信息（问题D2与D1的答案），右列对应报告者提供该信息的百分比（R1）。开发人员与报告者在第一项与最后一项依然一致，但是总的来说不一致性增加了。这里的斯皮尔曼等级相关系数是−0.03，表示开发者认为重要的信息与报告者提供的信息之间有巨大的差异。特别是报告者没有关注开发人员认为重要的信息。

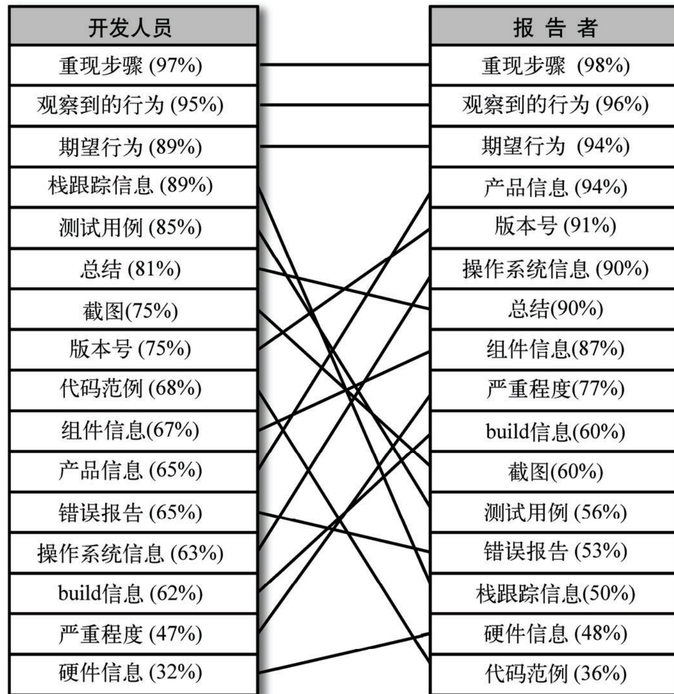


图24-3　开发人员使用的信息对比报告者提供的信息

有意思的是，图24-5显示了大多数报告者清楚开发人员的需求。换句话说，报告者之所以没有提供这些信息并不是因为他们不知道。与之前的图例相一致，左列表示开发人员眼中该信息的重要性，右列表示报告者眼中该信息的相关性（问题R3）。两部分基本上一致，只有截图一项出入较大。这里的斯皮尔曼等级相关系数是0.839，这也肯定了开发人员与报告者对于大部分信息的重要程度是有共识的。

总体而言，为了改进缺陷跟踪系统，人们可以告诉用户哪些信息是重要的（比如截图）。同时，系统应该提供更强大的工具来采集重要信息，因为这些信息用户常常很难获得。

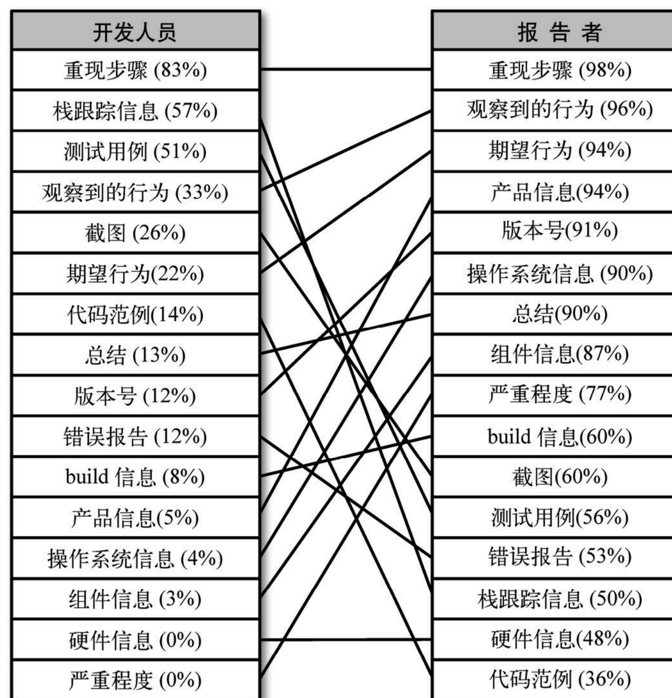


图24-4　开发人员认为最重要的信息对比报告者最经常提供的信息

24.5　缺陷报告的问题

在开发人员目前遇到的所有问题中，最常见的是信息不完整。其他普遍问题包括重现缺陷步骤有错误，测试用例出错，重复的缺陷，不正确的版本信息，观察到的行为和预期行为不准确。另一个让开发人员头痛的问题是报告者的语言熟练程度。这些问题都会在修复缺陷时轻易误导开发人员。

最严重的问题就是重现缺陷步骤中的错误以及不完整的信息。事实上，在D5问题中，很多开发人员都表示曾经因为缺陷报告中的信息不完整而受折磨。有一位开发人员这样写道：

> "导致延误的最大原因不是错误的信息，而是不完整的信息。"

我们对于垃圾函件出现的低频率并不是很意外，因为在Bugzilla与Jira中，报告者需要注册之后才能提交缺陷报告，注册过程成功地防止了垃圾报告。最后，栈跟踪信息出错的可能性非常低，因为它们大部分是直接复制粘贴至缺陷报告的，不过如果一旦出错，就会导致严重问题。

另一些重要问题在于测试用例与观察到的行为中的错误。有意思的是，重复的缺陷并没有十分困扰开发人员，尽管早期的研究认为它会是一个问题[1]。也许开发人员可以轻易地识别重复的缺陷，甚至得益于对同一问题的另一种描述。下一节将讨论重复缺陷报告的价值。

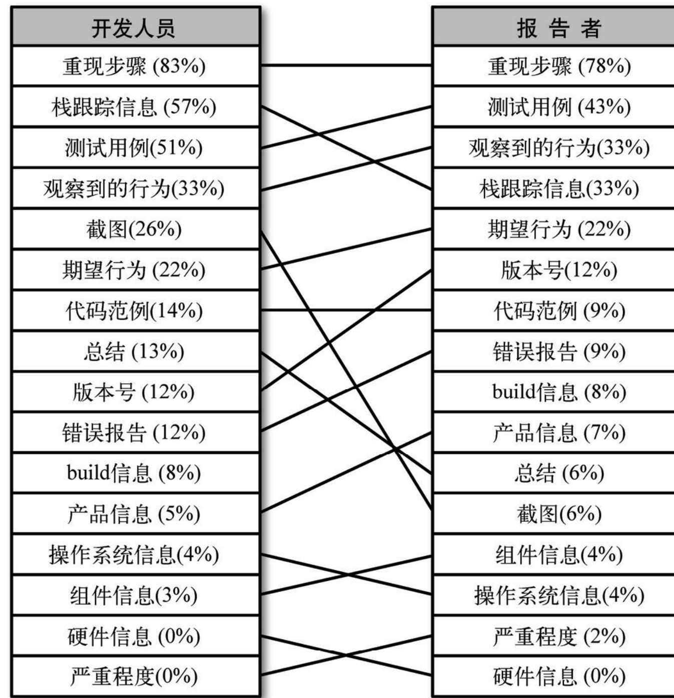


图24-5　开发人员认为最重要的信息对比报告者认为最重要的信息

24.6　重复缺陷报告的价值

常见的反对重复缺陷报告的观点是，它们会极大消耗缺陷跟踪系统的资源，并且需要质量保证团队投入更多的精力，而这些精力本可以投入到其他改进产品的工作中。本节中，我们会给出与这一观点对立的实证研究的证据：重复缺陷报告中其实含有额外的信息，有利于缺陷修复。

当一个缺陷报告被认为是重复的，我们通常的处理办法是将该缺陷关闭，并且信息会被删除。这种做法长期而言会对用户提交缺陷报告起到消极作用。当他们看到一个缺陷报告已经被提交过之后，他们不再愿意提供额外的信息：

> “通常，我报告的缺陷已经被能力不济的家伙提交过，而且报告得十分糟糕，缺少重要信息。当我花费很长时间编写的十分详尽的缺陷报告被标注是重复报告时，我感到十分沮丧……”

并不是所有人都认为重复缺陷报告是无益的。我们的调查发现有几位开发人员指出重复缺陷对于修复缺陷的益处：

> “重复缺陷报告并不一定就是麻烦。它们经常能提供额外的有用信息。虽然在另一

份缺陷报告中提交这些信息不是理想的做法。”

“最好能够修改报告，而不是注销一份很棒的报告只是因为在它之前已经有了一份差劲的缺陷报告。这样会对软件工程师更加有益。”

微软的一位测试构架师Alan Page有相似的观点，对于“为什么没有必要担忧重复的缺陷”他总结了三点原因[9]。

- 提交重复缺陷报告对用户通常都会有负面影响。他们会变得不愿报告缺陷，即使那个缺陷还没有被提交过。
- 鉴别人员比用户更擅长鉴别重复的缺陷，他们也更加熟悉系统。一个用户可能需要花费很长时间来浏览类似的缺陷，但鉴别人员只需要几分钟就能知道一个缺陷是否重复。
- 重复缺陷可以提供有价值的信息来帮助诊断出真正的问题。

重复缺陷真的提供了额外的信息吗？提供了多少呢？我们的实证研究将给出答案。首先，我们建立了一个工具来检测和量化信息，比如缺陷报告中的补丁、截图和栈跟踪信息。下一步，我们比较原始缺陷报告（称为主报告）和扩展报告中的信息。扩展报告中的信息涵盖了主报告与其他重复报告中的信息。

表24-2为我们在Eclipse项目中的发现。第一列为“信息条目”，指所有从缺陷报告中发现提取的信息。信息一共分为四类：事先定义的栏目（比如产品与部件信息）、补丁、截图和栈跟踪信息。第二列“主报告”指所有主报告中各类信息的平均数。比如，每一个主报告中只含有一个操作系统信息，在“主报告”一栏用1.000表示。这与我们很多的项目经验一致：当一个重复缺陷被发现之后，它就被轻易地关掉了，相关信息也被删除了。

第三栏“扩展报告”列出了在扩展缺陷报告中各类信息的平均数。当合并重复缺陷报告后，扩展缺陷报告中操作系统信息的平均数为1.631。这里合并了主报告与重复报告，所有的信息都被保留。第四栏“差异”指扩展报告相对于主报告的信息增长，是相对于单个主报告，重复缺陷报告增加信息的数量。例如，只要不删除重复报告中的信息，对于操作系统来说，重复报告就会增加平均0.631的信息。

表24-2的数据也表明大部分重复缺陷报告的提交者不是提交原始报告的人，这就解释了不同报告者数量的增长。

对于单个主报告，我们发现重复缺陷报告平均增加了0.113的补丁，0.062的补丁文件。6%的增长相对来说比较小，也表明了大部分补丁是针对主报告的，不过，重复缺陷增加了0.145个截图，使得截图的数量翻番。同时，重复缺陷也提供了额外的信息，比如版本、优先级与部件信息。

对于栈跟踪信息，我们比较了触发它们的异常情况以及前五个栈框架（stack frame）。Eclipse项目中，重复缺陷报告平均增加了0.918个栈跟踪信息（增幅182%）。其中，平均0.118个（61%）是发生于其他异常中，以及0.281个（115.2%）栈跟踪信息包含了没有被报告过的代码位置（在前五个栈框架中）。

这些发现表明重复缺陷提供了额外的信息，指向缺陷的起源，帮助开发人员修复它们。比如，面对程序崩溃时，更多的栈跟踪信息展现了更加灵活积极的方法，帮助人们找到嫌疑点。所以这

些发现也让我们重新思考缺陷跟踪系统中重复缺陷的存在价值。

表24-2　Eclipse中，相对一个主报告，重复缺陷报告增加的信息

信息条目	每个主报告的平均数量			
	主 报 告	扩展报告	差异[a]	
事先定义的栏目				
产品	1.000	1.127	+0.127	(12.7%)
组件	1.000	1.267	+0.287	(28.7%)
操作系统信息	1.000	1.631	+0.631	(63.1%)
报告的平台	1.000	1.241	+0.241	(24.1%)
版本信息	0.927	1.413	+0.486	(52.4%)
报告者	1.000	2.412	+1.412	(41.2%)
优先级	1.000	1.291	+0.291	(29.1%)
目标里程碑	0.654	0.794	+0.140	(21.4%)
补丁				
总数	1.828	1.942	+0.113	(6.2%)
单独补丁文件数量	1.061	1.124	+0.062	(5.9%)
截图				
总数	0.139	0.285	+0.145	(105.0%)
栈跟踪				
总数	0.504	1.422	+0.918	(182.1%)
独特异常	0.195	0.314	+0.118	(61.0%)
在最上层框架中的独特异常	0.223	0.431	+0.207	(93.3%)
在最上两层框架中的独特异常	0.229	0.458	+0.229	(100.0%)
在最上三层框架中的独特异常	0.234	0.483	+0.248	(106.4%)
在最上四层框架中的独特异常	0.239	0.504	+0.265	(110.9%)
在最上五层框架中的独特异常	0.244	0.525	+0.281	(115.2%)

a 所有信息条目的增长在P＜0.001处都很明显。

24.7　并非所有的缺陷都被修复了

大部分项目有许多缺陷因为资源或者时间限制而没有被修复。为了体现缺陷报告质量对于提高缺陷修复可能性的作用，我们从Apache、Eclipse和Mozilla项目中采样了15万个缺陷（每个项目5万个）。这些缺陷的结果分别为被修复、重复缺陷、转出、不予修复和未重现。我们将缺陷报告分为两组，成功的与失败的，然后使用统计测试比如卡方测定（Chi Square Test）与Kruskal-Wallis测试来检查缺陷报告成功与否与所提供信息（代码范例，栈跟踪信息，补丁，截图）之间的联系。我们通过如下指标来比较。

- 缺陷的结果

  我们比较了“被修复”的缺陷与其他状态的缺陷，比如“不予修复”和“未重现”的缺

陷。我们把重复缺陷报告作为单独的一组，因为有一些主缺陷报告已经被纠正过，另一些还没有。

- 缺陷的生命周期

  我们比较了生命周期较长与生命周期较短的缺陷。对于用户与开发人员来说，较短的生命周期是较为理想的。

我们也评估了可读性对于缺陷报告的影响。我们使用Style工具来度量缺陷报告的可读性，它“分析了文档撰写风格的表面特征”[5]。度量文档可读性就是计算每个字的音节数以及句子的长度。亚马逊公司使用文档可读性来评估书籍的阅读难度已告知读者，美国海军也用它来保证技术文档的可读性。在我们的实验中，我们使用了下列七种可读性指标：Kincaid、ARI（Automated Readability Index，自动易读性指数）、Coleman-Liau、Flesh、Fog、Lix与SMOG 等级。

实验结果如下：

- ❑ 缺陷报告中含有栈跟踪信息，被修复速度更快（Apache、 Eclipse和 Mozilla）；
- ❑ 缺陷报告更易于阅读，被修复速度更快（Apache、 Eclipse和Mozilla）；
- ❑ 缺陷报告中含有代码范例，被修复的可能性更高（Mozilla）。

并且，独立于我们的实验，Hooimeijer与Weimer发现Firefox中带有附件的缺陷报告会稍后才被处理，而带有很多注释的缺陷报告则会被更快地修复[8]。他们也确认了我们的发现即容易阅读的缺陷报告会更快被修复。Panjer观察到Eclipse项目中注释的数量，活跃度和严重程度对缺陷的生命周期最有影响力[10]。Schröter等人也佐证了我们的发现，即有栈跟踪信息的缺陷会更快被修复，并且重申了在缺陷报告中加入栈跟踪信息的重要性[11]。Guo等人在微软中做的一项关于Windows Vista与Windows 7的实验[7]中发现重赋值的数量，组织和地理分布，缺陷报告者声誉都会影响缺陷被修复的可能性。

本节中的发现表明，一份书写良好的缺陷报告帮助人们更好地理解缺陷，从而也能提高短时间内修复缺陷的可能性。

24.8 结论

缺陷报告对于维护软件产品质量至关重要。开发人员利用缺陷报告中的信息找出缺陷发生的原因，然后解决问题。但为了实现这一目的，要保证缺陷报告中的内容是可靠的，也是完整的。

但是一份高质量的缺陷报告是什么样的呢？我们从三个大型开源项目中对相关的开发者和报告者展开了调查，在本章中给出了分析结果，并从被访者的角度分析了缺陷报告中哪些内容是最重要的。开发人员的问卷调查结果显示他们认为栈跟踪信息，重现缺陷步骤，以及期望行为与观察到行为对于修复缺陷是十分重要的。有趣的是，来自缺陷报告者的调查结果与之类似，只是他们没有经常提供这类信息。这里开发人员所希望得到的信息与报告者实际提供的信息之间存在一个鸿沟。

所以我们可以通过改进缺陷跟踪系统来逾越这道鸿沟。比如建立自动采集数据的工具（自动获取程序崩溃时的栈跟踪信息），也可以在界面中提示报告者输入某些信息。我们的大量研究表

明重复的缺陷报告包含了一些其他有用信息，所以也许可以把新的信息自动合并入原始的缺陷报告中。同时我们还可以加入激励措施，比如列出提供某些信息的优点（有栈跟踪信息的缺陷报告修复得更快），鼓励缺陷提交者为改进缺陷报告多出一些力。

总体来说，缺陷报告质量越高，修复速度就越快。同时祝报告者与开发人员都工作顺利！

24.9　致谢

本章中的研究由我们和Nicolas Bettenburg、Sascha Just、Sunghun Kim、Adrian Schröter和Cathrin Weiss共同参与完成。

24.10　参考文献

[1] [Anvik et al. 2005] Anvik, John, Lyndon Hiew, and Gail C. Murphy. 2005. Coping with an open bug repository. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*: 35-39.

[2] [Bettenburg et al. 2008a] Bettenburg, Nicolas, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering: 308-318.*

[3] [Bettenburg et al. 2008b] Bettenburg, Nicolas, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate Bug Reports Considered Harmful...Really?. *Proceedings of the 24th International Conference on Software Maintenance*.

[4] [Breu et al. 2010] Breu, Silvia, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work:* 301-310.

[5] [Cherry and Vesterman 1981] Cherry, L.L., and W. Vesterman. 1981. Writing Tools—The STYLE and DICTION Programs. Computer Science Technical Report No. 91, Bell Laboratories, Murray Hill, NJ.

[6] [Goldberg 2010] Markham, Gervase, and Eli Goldberg. 2010. Bug Writing Guidelines. *https://developer.mozilla.org/en/Bug_writing_guidelines*.

[7] [Guo et al. 2010] Guo, Philip J., Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* 1: 496-504.

[8] [Hooimeijer and Weimer 2007] Hooimeijer, Peter, and Westley Weimer. 2007. Modeling bug report quality. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*: 34-43.

[9] [Page et al. 2008] Page, Alan, Ken Johnston, and B.J. Rollison. 2008. *How We Test Software at Microsoft*. Bellevue, WA: Microsoft Press.

[10] [Panjer 2007] Panjer, Lucas D. 2007. Predicting Eclipse Bug Lifetimes. *Proceedings of the Fourth International Workshop on Mining Software Repositories*: 29.

[11] [Schröter et al. 2010] Schröter, Adrian, Nicolas Bettenburg, and Rahul Premraj. 2010. Do Stack Traces Help Developers Fix Bugs?. *Proceedings of the 7th International Working Conference on Mining Software Repositories*: 118-121.

第25章

软件的缺陷都从哪儿来

Dewayne Perry

软件开发管理的终极目标是“更便宜，更快，更好”。可惜的是，很多时候目标虽然定得很好，但是实现目标的计划却差强人意。使软件“更便宜，更快”固然重要，但是对于很多软件系统来说，稳定性和安全性更为重要。在这种情况下，做出“更好”的软件才是最要紧的任务。

使软件更好的办法有很多。无论是清楚地认识客户的需求，还是最大限度地减少软件的故障等措施，都可以使软件更好。本章主要关注的是减少软件故障这一点。只有找到了问题的所在，我们才能对症下药地改进我们的产品或者流程。而监控故障相对来说是一种比较简单的方法，因为它要么本来就存在于项目的流程之中，要么就只存在于项目的回顾会议（常被称为“亡羊补牢会议”）里。

复杂性是软件系统最重要和最基本的特征，而控制这种复杂性则是减少软件故障的基本理念[6]。控制复杂性的方法有很多，最有用的一种是把软件组件的接口和实现区分开来。基于这个重要方法，本章将论述接口类故障和实现类故障的区别。

25.1　研究软件的缺陷

一个令人遗憾的现实是，关于软件故障的研究成果发表得很少，使得人们难以对软件故障的检测、改善和预防做更多系统的研究。不过从另一方面来说，项目组不愿发布这些研究成果也很正常，因为发布这些敏感的研究和数据，不仅可能引起组织内部的矛盾，也会带来外部的竞争压力。

虽然研究软件故障的资料很少，我们仍然有一系列里程碑式的关于软件故障的研究，可以作为改善产品和流程的基础。Endres[8]、Schneidewind及Hoffman[26]以及Glass[10]都发表过这方面的论文。不过，他们的研究有着一个共同的缺憾，就是没有把接口故障作为一个专门的分类进行研究。

Thayer、Lipow 和 Nelson [29]以及Bowen[5]在他们的著作中非常详尽地阐述了软件故障的分类，但是关于接口故障的部分却相对局限。Basili和Perricone[2]曾发表过极为全面的软件故障研究报告。报告集中讨论了中等大小的软件系统在初始开发阶段遇到的问题，他们记录下了故障的数据、

受影响的组件数目、故障的类型以及解决故障所需的工作量。其中，接口故障占据了39%，是数量最多的故障类别。

不过我们发现，这些研究中，没有一个提到关于超大型软件系统中可能遇到的问题，也没有提到在软件的后续开发阶段中遇到的问题。Perry 和Evangelist[19][20]是首先研究大型实时系统后续开发阶段故障的人。他们的研究成果中有一项极为重要，那就是接口故障是目前为止最常见的故障，占到所有故障的68%。这就留下了一个问题：这些接口故障是否容易检测和修复？

要知道，软件的初始开发和后续开发是截然不同的。对于前者，设计和实现工作的限制很少；而对于后者，由于开发人员常常必须基于已经存在的系统做出修改，所以不但限制很多，而且要理清楚修改相关的连带关系也变得越发困难。软件的后续开发阶段比初始开发阶段要长得多，所以关于这个阶段出现的软件故障的研究也重要得多。

在这一章里，我们将详细研究一个超稳定、超大型的实时软件系统的一个发行版本。之所以没有把多个来自各行各业的中型系统拿来做横向研究，是因为如果只研究一个大型的系统的话，我们可以对整个系统认识得更深，也对其问题认识得更深。当然缺点也显而易见，那就是研究成果相对来说没那么容易适用于更多的情况。这种利弊的权衡在实证研究中很常见。

接下来我们将看到的，是这种深入的认识带给我们的一些有用的见解。比如，我们常常认为“我们只要能发现bug，就可以很容易地修复它”。但实际上，我们的数据和这样的“常识”却正好相反。此外，对于之前留下的那个问题，得到的数据支持了原来的猜测，即：接口故障要比实现故障更难以修复。

25.2　本次研究的环境和背景

本章所讨论的这个系统是一个非常大的（100万行代码或更多）分布式实时系统，主要用C语言写成（也包含部分针对不同领域的程序语言），系统的开发基于Unix环境，多个工作站分布于不同的地点同时进行开发。

项目组织结构是典型的大型项目结构，由多个不同组织提供人员，每个组织负责不同部分的开发。各个组织分别负责需求分析、架构、设计、编写代码、机能测试、系统及稳定性测试以及α测试。

开发流程也是典型的大型项目开发流程。系统工程师先准备非正式的结构化文档，定义修改系统的需求。然后设计师们根据这些需求准备非正式的设计文档，再根据该设计的大小选择3～15名人员进行正式的评审。然后这些设计将被分为小块进行底层的设计和代码编写。在这个阶段，将有3～5个评审员对产品进行评审，并有人员对其进行底层单元测试。当组件通过这一系列程序，可以投入使用时，开发人员将不断进行整合和系统测试，直到系统整合完成。

我们讨论的这个版本是一个“后续开发版本”。我们可以把这样的一个版本看做是这个系统发展过程中的一个节点。由于规模很大，这个系统的发展包含着多个并行的版本。也就是说，虽然版本发表的日期是按照顺序来的，但每个组件都是同时在开发，都处于不同阶段，而且都有着自己的版本。这种并行性带来的是版本之间的依赖性，以及由此引发的更多的问题。除了版本的

并行性以外，每一个新版本中增加的代码量（大约每个版本15%～20%的新代码）和变更（bug修改、功能改进、新功能等）都大同小异。正是基于这两点，我们认为研究中的这个版本可以作为整个软件生命周期中的一个代表性的节点。

测试环节中发现的故障会被报告并由一个修改请求（Modification Request，MR）跟踪系统（例如 CMS [25]）来监控。只有通过这个系统才能修改源代码。这就保证了所有的修改（无论是故障修复，添加功能或者是改善功能）都会被自动记录下来。不过需要留意的是，故障跟踪只会在测试和发布环节进行，并不会在构架、设计和代码编写部分进行。在这些环节出现的问题不会被MR系统记录，而是由开发人员自行解决。

这个研究的目标是通过专注分析一个特定系统的某一个版本，深入了解当前的开发流程。我们使用的方法是问卷调查。我们准备了一份问卷，发给那些解决故障的开发人员。我们首先调查所有类型的故障，再深入分析这些故障中发生率最高的类型。虽然以前有过用随机问卷调查的小型研究，但这种类型的问卷调查尚属首次。我们先用MR系统的故障数据库来确定我们需要进行调查的故障和解决这些故障的开发人员，然后再将这些问卷发给他们。

由于种种原因（包括时间安排方面的压力），项目管理层给我们的研究做了很多的限制：第一，我们绝对不能妨碍到项目的正常进行；第二，研究对象必须是自愿参与；第三，研究必须是完全匿名的。我们的研究受到这些限制的制约将不可避免地有一些缺陷。本章后面部分将会讨论这些限制对此次研究的有效性的影响。

这就是本次研究的背景。所有问卷调查、分析以及结论都基于此背景。

25.3 第一阶段：总体调查

总体调查有三个具体的目的：

- 确定我们找到了何种问题（我们将在此讨论），以及，在开发这个版本的过程中，遇到了何种只跟这个应用程序相关的问题（我们将不会讨论，因为没有普遍性）；
- 确定问题是如何被发现的（即在哪个测试的环节被发现）；
- 确定问题是何时被发现的。

在经验性问卷调查中我们常常遇到的一个问题，就是如何保证这些问卷都是用被调查者的“语言”来写的。这里的“语言”不止包括公司内部或者项目组内部的专用术语，也包括专用的流程。在做调查的时候，我们希望接受问卷调查的开发人员能够完全地理解所问到的问题。如果被调查者看不明白问题，将会使研究的准确性受到质疑。为了解决这个问题，我们请了开发人员帮助我们设计这个问卷，我们也使用了项目组的术语和流程以便于被调查者理解问卷上的问题。

25.3.1 调查问卷

第一阶段的问卷包含了两个主要的部分：一是确定故障的分类，二是确定发现故障的具体测试阶段。确定故障的分类有两个重点：一是引入故障的阶段，二是故障的类型。由于在总体调查阶段所调查到的故障类型常常只跟这个程序或者这个项目使用的方法相关，所以我们更侧重于故

障起源方面的特质。下面我们列出故障的分类。

- 遗留

  前一个版本未解决而遗留下来的问题。
- 需求

  在需求分析环节出现的问题。
- 设计

  在构架和设计环节出现的问题。
- 代码编写

  在代码编写环节出现的问题。
- 测试环境

  在搭建或者准备测试环境时出现的问题（例如系统设置错误，静态数据等）。
- 测试

  测试中发生的问题（如运行故障等）。
- 重复

  已经报告过的问题。
- 非问题

  由于用户误解界面或者功能所产生的问题。
- 其他

  其他无法分类的问题，例如硬件问题等。

问卷的另一个组成部分是关于发现故障的测试环节。测试分为以下几类：

- 性能测试（Capability Test，CT）

  分离系统的不同部分并确认每个部分性能的测试。
- 系统测试（System Test，ST）

  在实验室环境中确定系统整体运行正常的测试。
- 系统稳定性测试（System Stability Test，SS）

  在实验环境中，模拟长时间负荷运行的测试。
- α 测试（Alpha Test，AT）

  由友善的用户（通常为组织内部人士）对即将正式发行的版本进行使用测试。
- 发布后测试（Released，RE）

  即正常使用；但需要注意的是，在我们的研究中这一项指的不是当前的版本，而是上一个版本的使用；我们希望这项数据能够让我们预见到当前版本的问题。

除此之外，我们还从MR数据库中提取了这些故障从发现到解决所花费时间的数据。

在理想状态下，这些测试都是按顺序进行的。但实际上当在开发规模较大和复杂程度较高的系统时，这些测试环节可能有所重叠。我们分析一下这种重叠的成因。首先，系统的多个部件同时在被修改。这就意味着在任意一个时间点上，这些部件都分别处于不同的开发阶段。其次，软

件开发的反复性使得我们在开发某些部件的时候常常退回到之前的阶段。第三，即使知道组件不完整，我们仍然需要尽早进行测试以发现可能存在的问题。这样看来，测试遵循一种从下至上的层级模式：先是测试各个部件，再测试多个部件组成的子系统，最后将测试好的子系统整合成为完整的系统。某个部件该在什么时候进行下一阶段的测试通常由开发人员自己判断，而多数情况下这取决于功能的完成度和已经进行了的测试数量。我们也可以说，系统规模和复杂程度越大，系统负载和压力就越大，测试的阶段也就越后期。

25.3.2 数据的总结

表25-1总结了这些故障的类别情况。发生在初期的故障（即需求、设计或者代码编写环节）最多，约占总数的33.7%。由于在通常情况下设计类故障和代码编写类的故障难以分别，我们决定把它们分作一类：即设计/代码编写类故障，占总数的28.8%。不过，在这个项目所使用的流程中，需求类和设计/代码编写类的故障之间的区别是很明显的：系统工程师写需求规格文档，开发人员做设计并编写代码。

表25-1 故障总结

类　型	比　例
遗留	4.0%
需求	4.9%
设计	10.6%
代码编写	18.2%
测试环境	19.1%
测试	5.7%
重复	13.9%
非问题	15.9%
其他	7.8%

紧接着，下一个出现较多的故障类型是和测试相关的故障（包括测试环境和各测试阶段中找到的故障），占总数的24.8%。要对这样庞大和精密的实时系统进行测试，需要在试验室中模拟出接近现实世界的复杂运行环境，出现这么多问题也不奇怪：首先，测试环境本身就是一个庞大而精密的系统，它本身就需要做很多测试；其次，随着实时系统发展得越来越大越来越复杂，测试环境也必须跟着发展起来。不过总的来说，这是一个普遍存在的问题，它需要更深入的研究来解决。

“重复”和“非问题”类故障是另一个大头，占了总量的29.8%。传统观念中常将这两个看作是非必要开销（overhead）的一部分。确实，出现大量重复类故障是由大型项目中各种工作的并发性所引起的，所以难以避免。然而，大部分非问题类故障却是因为不严谨或者过期的文档所带来的误解造成的。不过可以肯定的是，为减少这两类问题而采取的各种措施也会对其他问题的解决起积极作用。无论如何，减少不必要的行政开销，都将提高项目的成本效益。

“遗留”类故障的数量代表着在这样一个大型的实时系统中发现问题的困难程度。这些问题可能在上一个版本中根本没能被察觉，直到后来系统的功能发生变化之后才凸显出来。

图25-1展示了不同测试阶段中的需求、设计及代码编写类故障。我们主要关注了软件开发早期的故障，因为这个阶段的故障最多，更多的关注可以让我们得到更多的数据。

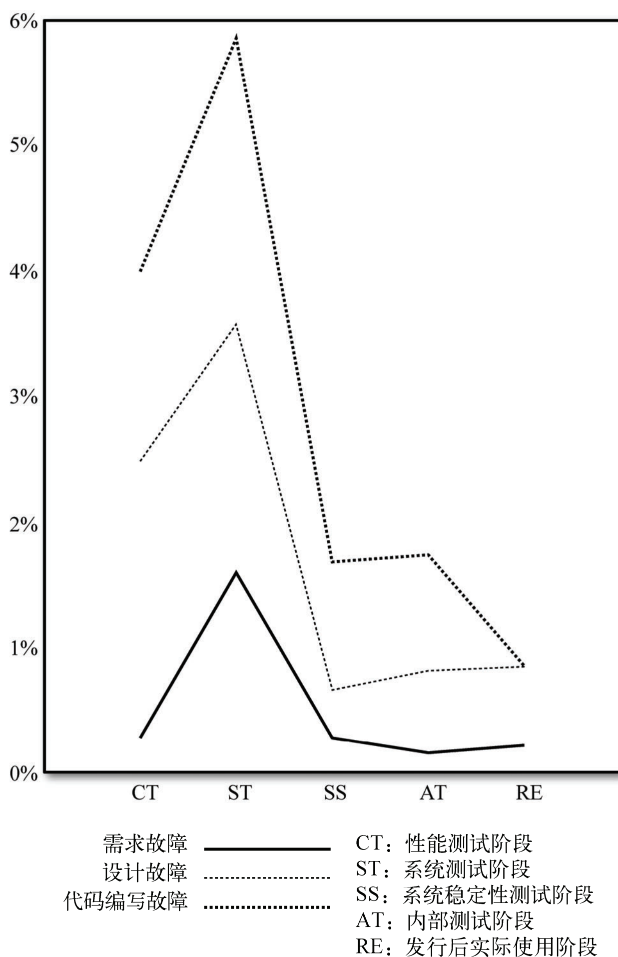


图25-1　故障分类图示（按阶段）

请注意图25-1和图25-2中的故障百分比是相对于所有故障的，而不是相对于图中所显示的故障。此外，在图25-1中各个测试环节呈顺序排列，但实际上，在几乎所有软件系统的开发过程中，各个环节相互间存在着很多并行和重叠的情况。虽然软件开发的流程由一系列（通常是迭代）的环节组成，但由于成百上千的工程师同时在开发成百上千个功能，实际的开发流程和我们所熟知的瀑布模式完全不同。这也正是为什么我们用了另一幅图25-2来按时间顺序展示和图25-1同样的数据。

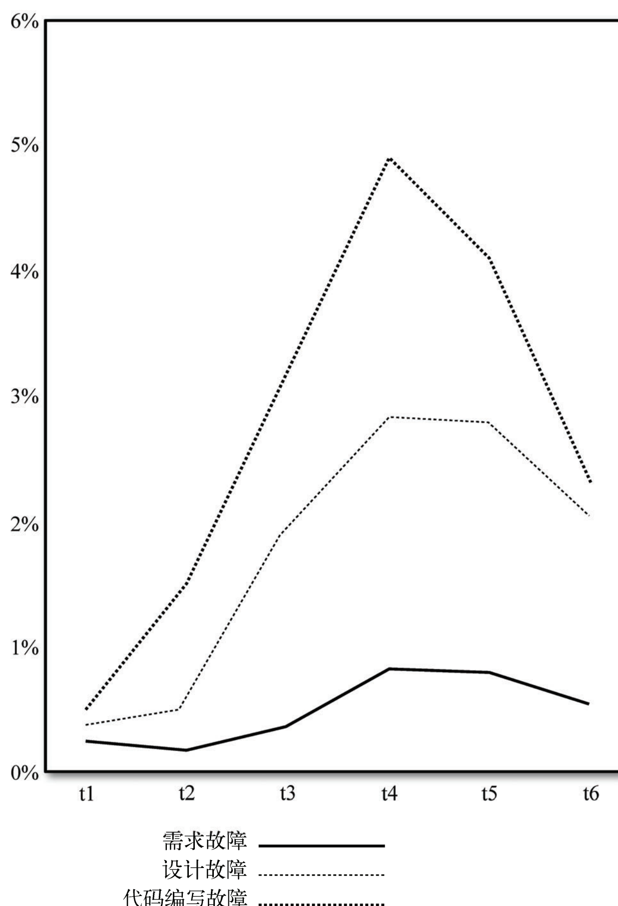


图25-2　故障分类图示（按时间）

有两个重点值得留意。首先，系统测试（ST）是各类故障高发的环节，性能测试（CT）紧随其后。这并不奇怪，在这两个环节中我们通常做好了迎接问题的准备，并会费尽心思找出问题。在系统测试的环节中，所有组件第一次被整合成系统，所以极可能出现很多部件之间不吻合或者配合不当的问题。也就是说，虽然所有测试的目的都是为了找到错误，但是在系统测试中我们会尤其小心翼翼，试图找到更多的错误。而性能测试和单元测试寻找的故障是类似的——都是为了找出内部和初始交互中的故障。这种特性限制了在性能测试中能找到的故障类型。

我们观察到的第二点，是在每一个测试环节中都发现了所有类型的故障。例如代码编写类的故障在每个测试环节中都有发现。这并不奇怪，一个显而易见的解释是早期的测试环节中发现的问题需要后来不断地修改系统才能解决。此外，虽然从头到尾都能见到设计和需求类的故障让人感到很担心，但是我们认为这是由于需求/设计文档不够准确和完整而造成的。而且，我们认为这是一个普遍存在于流程层面的问题，而不是仅存于这个项目中的问题。

图25-2中的时间点是均匀分布的。从图的形状来看，我们不难看出系统测试和t4区间之间的

重叠。很不幸的是，我们只有时间数据（即故障的发现和解决时间），如果有工作量数据[15]（即从故障的发现到解决中实际花费的时间）作为参考对比将使我们的研究更有意义。

此外，我们从图25-2中还能看出，几乎所有类型的故障，包括需求、设计和代码编写类故障的数量都在t4这个点达到了峰值，并且几乎都保持到了t5。除了代码编写类故障，它从t4开始慢慢减少。

25.3.3　第一阶段的研究总结

下面这些是我们从研究这个不断成长的大型实时系统中观察到的一些结果。

- 代码编写类、测试类和行政上的非必要开销所占的比例大致相同。
- 需求问题虽然数量并不非常大，但却非常重要（尤其因为大部分需求问题都是在晚期的测试中才被发现）。
- 在每一个测试环节中都发现了所有类型的故障。
- 在强度最大的测试环节（即系统测试环节）发现的故障最多。
- 多数故障都是在测试周期的后期才被发现。

这些总结有一定局限性，因为追踪错误MR主要是一个测试行为。如果我们可以观察记录在项目开发早期出现的故障类型和数量，那将对我们的研究非常有用。此外，如果可以将检测需求和设计类故障的方法也算入开发流程的一部分，也会对研究有帮助。

25.4　第二阶段：设计/代码编写类故障调查

基于总体调查的结果，我们决定深入调查设计/代码编写类故障。在这个部分的研究中，我们想要达到的目的如下：

- 确定发生在设计和代码编写中的故障的类型；
- 确定找出故障、再现故障、解决故障的难度；
- 确定出现故障的深层次原因；
- 找出可能避免故障的方法；
- 对比寻找和解决接口类故障和实现类故障的难度。

选择这个类别的故障作为这个部分的研究对象有两个原因。第一，要区别这两种故障实在非常困难。第二，在项目早期发现这两种故障将会极大地降低花费在发现故障上的总体成本，也就是说，在系统整合之前发现故障比在整合之后的实验室测试阶段再发现故障所需的成本要小得多。在这一点上，我们得到的数据和Boehm的研究数据[4]一致（参见Barry Behem所作的第10章）。

在下面两个小节，我们将分析我们使用的调查问卷，提出我们的统计分析结果，以及我们关于接口和实现类故障的研究结论。

25.4.1　调查问卷

我们让参加调查的人员指出找到和解决故障的难度，确定出现故障的深层次的原因，指出最

好的预防或者避免这些问题的方法，并告诉我们他们对自己的回答有多大把握。需要注意的一点是，参与调查的人是在最后解决故障的人。具体问题如下所示。

- 按照发现和重现的困难程度为每个故障分级。
  (1) 容易——可随意重现。
  (2) 中等——偶尔（间歇性）可以重现。
  (3) 困难——需要推测和尝试才能找出重现的方法。
  (4) 非常困难——极难重现。

- 要解决这个故障需要多少时间来设计和编写代码，以及用文档记录和测试的时间。（注意在单程序员的开发环境中能很快解决的问题，在多人同时作业，复杂的实验室测试开发环境中将需要更多时间来解决）
  (1) 容易——1天内即可解决。
  (2) 中等——1到5天内。
  (3) 困难——6到30天内。
  (4) 非常困难——30天以上。

- 对于每个故障，考虑以下22个原因并从中选择一个作为直接造成故障的原因（即故障的类型）。
  (1) 程序语言陷阱——如指针问题，或者“=”代替了“==”的问题。
  (2) 通信协议——未按照进程间通信协议约定造成问题。
  (3) 底层逻辑——例如无限循环问题，指针初始化问题，等等。
  (4) CMS复杂度——例如由软件变更管理系统（Change Management System，CMS）的复杂度引起的问题。
  (5) 内部功能性——模块/子系统内部功能不完整，或者需要添加、修改。
  (6) 外部功能性——模块/子系统外部功能不完整，或者需要添加、修改。
  (7) 原始数据类型误用——设计或者代码编写需要依赖的原始数据类型未被“正确使用”。
  (8) 原始数据类型不支持——设计或者代码编写需要依赖的原始数据类型尚未开发完全（即原始数据类型不能正常使用）。
  (9) 变更协调——不知道之前的变更或者依赖同时进行的变更。
  (10) 接口复杂度——接口的结构差或者过于繁杂。
  (11) 设计/代码复杂度——实现层面的结构差或者过于繁杂。
  (12) 错误处理——对于异常的处理或者修复不正确。
  (13) 争用情况——数据的共享没协调好。
  (14) 性能——如不能满足程序在实时性，资源的访问效率或者反应时间上的规定。
  (15) 资源分配——资源的分配或者解除分配不当。
  (16) 动态数据设计——动态数据资源或者结构的设计不当。
  (17) 动态数据使用——动态数据资源或者结构的使用不当，例如初始化或者保持约束条

件（constraints）不当等。

(18) 静态数据设计——静态数据结构（例如存放位置，分区和冗余）设计不当。

(19) 未知交互——与其他功能或者其他系统的组件之间的未知牵扯。

(20) 意外依赖——对其他系统组件意料之外的依赖性或者和其他组件的意料之外的互动。

(21) 并发工作——对其他版本中正在做的工作有意料之外的依赖性。

(22) 其他——请描述该故障。

- 如果故障是一个病症，请尝试找出诱发这个这个病症的深层原因。

(1) 未给出——未能给出深层的原因。

(2) 不完整/遗漏的需求——由于不完整或者未声明的需求造成了故障。

(3) 模棱两可的需求——需求的声明不严谨，可以解读出不同的意思，最后没能按照正确的意思来实现。

(4) 不完整/遗漏的设计——由于不完整或者未声明的设计规范造成了故障。

(5) 模棱两可的设计——设计规范不严谨，可以解读出不同的意思，最后没能按照正确的意思来实现。

(6) 早前修复不当——故障是在解决早前的错误时处理不当造成的（即故障并非由新的开发造成）。

(7) 认识不足——有些东西本来是需要我来了解的，但我没意识到。

(8) 错误的修改——我怀疑提交的故障解决方案有误，但是我不确定怎样来正确地解决它。

(9) 被迫提交方案——由于受到压力（通常是来自于时间安排上的压力）而不得不在明知解决方案有误的情况下强行提交。

(10) 其他——请阐述原因。

- 对于这个故障，考虑可能的预防措施，并在下面的列表中选择一个最有用或者最合适的。

(1) 更规范的需求——用正规的表示法（可以是文字或图示）精确而清楚地表述需求（或者设计）信息。

(2) 提供需求/设计模板——提供更详细的需求/设计文档模板。

(3) 更规范的接口规格——使用正规的表示法来描述模块接口。

(4) 提供培训——提供研讨会、讲座以及正规培训课程。

(5) 进行程序演练——大致确定程序进程及数据对象间的互动。

(6) 提供专业人士/文档——当需要时提供一位专业人士或者清晰的专业文档。

(7) 使设计/代码编写同步——保证设计文档和当前代码的一致性。

(8) 严格执行准则——严格执行代码审查以及使用静态代码分析工具，如lint。

(9) 更好的测试安排——提供更好的测试计划及更好地进行执行（如自动回归测试）。

(10) 其他——请说明其他预防的办法。

被调查者为自己的回答的自信度打分，按顺序分别是：非常高，高，中，低，非常低。我们排除了少量自信度为低或者非常低的调查结果。

25.4.2 统计分析

我们收到了68%的问卷回复，并排除了6%的“低”或者“非常低”自信度的回答。对剩余的问卷，我们使用卡方（Chi-square）分析法[28]来测试各个配对数据组之间的独立性（或者相关性）。在用卡方分析法中，χ^2 越低就代表着数据相互间越独立，越高则代表越相互依赖。而p值则代表数据的显著性：p值越低，那么我们分析出的结果纯属巧合的可能性就越小。在表25-2中我们可以看到，预防措施和故障寻找难度的联系最不紧密（他们的χ^2值最低），而且这种不紧密性有很高的显著性（p值低于0.05代表着只有不到1/20的几率这种相互独立的关系是因为巧合）。故障类型和故障成因，故障类型和故障预防，还有故障成因和故障预防这三对数据的相互关联性最为紧密。我们观察到他们的χ^2 值是本次调查中最高的三个，而且显著性非常之高（只有不到1/10000的几率是因为巧合）。

这三对数据的紧密联系是好事，因为：(1) 故障确实应该和其成因密切相关；(2) 故障类型和成因也确实应该和预防措施紧密相关。这表明接受问卷调查者的回复是前后一致并合乎逻辑的。

表25-2 卡方分析总结

数 据 组	自 由 度	χ^2	p
发现，解决	6	51.489	0.0001
类型，发现	63	174.269	0.0001
类型，解决	63	204.252	0.0001
成因，发现	27	94.493	0.0001
成因，解决	27	55.232	0.0011
类型，成因	189	403.136	0.0001
预防，成因	27	41.021	0.041
预防，解决	27	97.886	0.0001
类型，预防	189	492.826	0.0001
成因，预防	81	641.417	0.0001

1. 发现和解决故障

表25-3展示了发现及解决设计/代码编写故障的难度。其中，78%只需5天以下即可解决。总的来说，容易找到的故障就比较容易修复，越难找到的故障就越难解决。

表25-3 找出和解决故障的时间对比

找出/解决		< 1天	1～5天	6～30天	> 30天
		30.1%	**48.8%**	**18.0%**	**3.6%**
容易	**67.5%**	23.7%	32.1%	10.0%	1.7%
中等	**23.4%**	4.2%	12.5%	5.6%	1.1%
困难	**7.7%**	1.7%	3.4%	2.1%	0.5%
非常困难	**1.4%**	0.5%	0.3%	0.3%	0.3%

卡方分析法的一个有趣之处是：分析结果是基于配对数据的“预期值”和观察到的“实际值”之间的差别。在这次的分析中，预期值是“找出”的百分比和“解决”百分比的乘积。如果这两个数据互无关系，那么我们的预期值将等于或者非常接近于观察到的实际值，否则这两组数据便有关联。

第一行的数据是我们观察到的故障解决时间的百分比，而第一列是找出和重现故障的难度的百分比。故障是“容易找出”并且“可以在1天内解决”的预期可能性是67.1% ×30.1%＝20.2%，而实际观察到的结果是23.7%，比预期值多了17%。

观察到的结果是容易找出并可以在1天内解决的故障比分析出来的预期要多。但有趣的是，故障是“容易找出”并“需要6～30天解决”的实际比率（10%）却比预期比率（12%）要少。

虽然表25-3中不同的“发现故障的难度”和“解决故障的时间”的数据相互之间不具备可比性，但我们还是发现了一个耐人寻味的关联。把表中的多项数据合并之后我们发现了一个有意思的现象，在表25-4中体现。这似乎正好推翻了那句老话“问题一旦找到了，解决起来就容易了”。有很大一部分找出难度是“容易/中等”的故障需要较长的时间来修复。

表25-4　寻找/解决难度总结表

找出/解决难度	≤5天	≥ 6天
容易/中等	72.5%	18.4%
困难/非常困难	5.9%	3.2%

2. 故障

表25-5列出了所有故障类型，并按出现频率排序。为简洁起见，我们将在接下来的表中用编号来表示故障类型。

前五种故障占到了总数的67%。“内部功能”类故障的数量遥遥领先，让人有些意外，但“接口复杂度”是一个严重的问题却是在意料之中。不过总的来说，这样一些故障排在前列还是合乎这个系统的发展特性的。在发展中，系统被加入了大量的新功能，很容易引发“内部功能”、“底层逻辑”及“外部功能”类故障。

这个实时系统规模巨大且极其复杂，所以我们不难理解在开发中为什么会出现“接口复杂度”，“意外的依赖性”，“设计/代码编写的复杂性”，“变更协调”和“并发的工作”等故障。

C语言有一些为人熟知的“语言陷阱”，造成了列表中游的某些故障。比如“争用情况”就是因为C语言不具备某些语言特性而造成的严重问题。

“性能”相对来说不是一个重大的问题。这大概是由于我们分析的这个版本已经是一个成熟的发行版本的原因。此外，代码检查的时候也会着重注意检查性能问题，这样性能所引发的故障就不会很多了。

3. 用工作量权衡后的故障频率

在对故障排序的时候，有两个重要的问题需要我们考虑：故障的检测难度对排序的影响，以及故障的解决难度对排序的影响。给列表中的故障类型加上权重可以让我们用故障的检测及解决

难度来修正我们此前的排序。仅从“不浪费任何数据”的角度来看，故障的频率确实可以作为一个判断其重要性的初级指标。表25-5列出了各个故障类型，并按照频率排列。

表25-5 故障类型（按频率排列）

故障类型	频率 %	故障描述
5	25.0%	内部功能
10	11.4%	接口复杂度
20	8.0%	意外的依赖性
3	7.9%	底层逻辑
11	7.7%	设计/代码复杂度
22	5.8%	其他
9	4.9%	变更协调
21	4.4%	并行的工作
13	4.3%	争用情况
6	3.6%	外部功能
1	3.5%	语言陷阱
12	3.3%	错误处理
7	2.4%	原始数据类型误用
17	2.1%	动态数据使用
15	1.5%	资源分配
18	1.0%	静态数据设计
14	0.9%	性能
19	0.7%	未知的交互
8	0.6%	原始数据类型不支持
2	0.4%	协议
4	0.3%	CMS复杂度
16	0.3%	动态数据设计

在表25-6中我们尝试将不同故障的检测难度用权重表示出来。计算权重的方法是将难度从易到难分为1、2、3、4，分别乘以该难度所对应的百分比，再汇总。

很明显，如果我们有更详细的故障检测所花费的时间数据，并按照该数据调整权重值（正如接下来我们分析故障修复时间数据时要做的），将会更好。也就是说，我们所算出来的检测难度权重值具备参考性，但不能单独依靠这些数据得出结论。

我们尝试了多种不同的权重方案，结果大同小异，所以我们选择了这个最简单的方案。

如果对于每一个故障都有相关的花费时间数据，那我们便能对其难度有更切实的感受。但这类数据很少见，所以我们只能用一些估计的数据来代替。

举例来说，如果对于某个种类故障的检测，66%属于容易发现，23%中等，11%困难，0%非

常困难，那么其权重即是：145=(66 × 1)+(23 × 2)+(11 × 3)+(0 × 4)。表25-6展示的是经过了权衡以后的各种故障的检测难度，按从最简单到最困难的顺序排列。

表25-6　故障检测难度权重值一览

故障类别	百分比（易/中/难/很难）	权　重	故障描述
4	100/0/0/0	100	CMS复杂度
18	100/0/0/0	100	静态数据设计
7	88/8/4/0	120	原始数据类型误用
2	75/25/0/0	125	协议
20	78/16/5/1	129	意外的依赖性
21	70/23/2/4	130	并行的工作
3	73/22/5/0	132	底层逻辑
22	82/12/2/5	132	其他
5	74/19/6/1	134	内部功能
6	67/31/3/0	139	外部功能
1	68/26/2/2	141	语言陷阱
10	66/23/11/0	145	接口复杂度
9	65/20/12/2	149	变更协调
8	67/17/17/0	152	原始数据类型不支持
19	88/8/4/0	157	未知的互动
16	67/0/33/0	157	动态数据设计
17	52/38/10/0	158	动态数据使用
15	47/47/7/0	162	资源分配
12	55/30/12/3	163	错误处理
11	55/29/16/1	165	代码复杂度
14	56/11/11/22	199	性能
13	12/67/21/0	209	争用情况

一般来说，“性能”和“争用情况”这两种问题比较难以分离和重现。可以预见的是，“代码复杂度”和“错误处理”类的故障也会很难检测和重现。此外，“语言陷阱”和“接口复杂度”类故障都比较难检测。

从卡方分析法的结果看来，“内部功能”、“意外的依赖关系”和“其他”类的故障往往比预想的要容易检测，而“代码复杂度”和“性能”类故障则常常比预想的更难检测。当被调查故障的数量增大，预期和实际的差异也就变得越来越明显。

如果我们用刚刚算出的权重去权衡表25-5中的频率数据，故障的排序并不会有什么大的变化。只有“内部功能”、“代码复杂度”和“争用条件”（编号分别为5，11，13）三类故障的顺序稍微有些改变。

表25-7列出了用实际花费时间来权衡之后，解决各类故障的难度。我们用1，3，15，30作为平均时间值，并以此作为权重，而计算方法则仍按照此前计算故障检测时间的权重的方法。

在用解决故障的时间做权重之后，故障的排序起了一些有趣的变化。“语言陷阱”，“底层逻辑”和“内部功能”类故障的排名剧烈下降。这和我们对这几类故障的直观感觉正好一致。“设计/代码复杂度”、“资源分配”和“意外的依赖性”类故障（编号11，15，20）的排名强力上升。“接口复杂度”、“争用情况”和“性能”类故障（编号10，13，14）的排名也有一定上升。

表25-7　故障修复难度权重值一览

故障类别	百分比（易/中/难/很难）	权　重	故障描述
16	67/33/0/0	166	动态数据设计
4	67/33/0/0	166	CMS复杂度
8	50/50/0/0	200	原始数据类型不支持
18	50/50/0/0	200	静态数据设计
1	63/31/6/0	244	语言陷阱
3	59/37/3/1	245	底层逻辑
2	25/75/0/0	250	协议
17	38/48/14/0	392	动态数据使用
9	37/49/14/0	394	变更协调
5	27/59/14/0	414	内部功能
22	40/43/12/5	496	其他
7	46/37/8/8	497	原始数据类型误用
10	17/57/26/1	608	接口复杂度
21	25/43/30/2	661	并行的工作
6	22/50/22/6	682	外部功能
13	16/56/21/7	709	争用情况
12	21/52/18/9	717	错误处理
19	29/43/14/14	785	未知的互动
20	24/39/33/5	786	意外的依赖性
11	22/39/27/12	904	设计/代码复杂度
14	11/22/44/22	1397	性能
15	0/47/27/27	1356	资源分配

表25-8列出了经过权重之后的，最难解决的四种故障。根据我们的权重方案，这四种故障消耗了55.2%的故障解决时间，以及51%的故障检测时间，从数量上来说占了总数的52.1%。总的来说，这些故障较容易找出但是较难解决。虽然故障的“检测难度”和“修复难度”这两个难度严格来说并不具备可比性，但发现这样的一致性还是很有趣的。

表25-8　最难修复的故障（权重后）

故障类别	权重后的%	故障描述
5	18.7%	内部功能
10	12.6%	接口复杂度
11	12.6%	代码复杂度
20	11.3%	意外的依赖性

从卡方分析的结果来看，“语言陷阱”和“底层逻辑”类故障比起预期需要更少的时间来解决。“接口复杂度”和“内部功能”类故障比起预期更经常在1到6天内解决，而“设计/代码编写”和“意外的依赖性”带来的故障常需要更多的时间（即6到30天）来解决。这些偏差巩固了给故障修复难度加上权重的重要性。

4. 深层的诱因

在表25-9中，我们列出了各类故障的诱因，并把它们按照未经权衡的频率排列。

表25-9　故障的深层诱因

故障类别	频率 %	故障描述
4	25.2%	动态数据设计
1	20.5%	未给出
7	17.8%	认识不足
5	9.8%	模棱两可的设计
6	7.3%	早前修复不当
9	6.8%	迫于压力提交
2	5.4%	不完整/遗漏的需求
10	4.1%	其他
3	2.0%	模棱两可的需求
8	1.1%	错误的修改

对于“未给出”分类的高频率我们有必要解释一下。其一是因为“语言陷阱”、“底层逻辑”、“争用情况”和“变更协调”这些既是故障的类型又是故障发生的原因，所以在选择这些的故障的深层诱因时，答案通常会是“未给出”（在未给出原因的故障中有7.8%属于这类故障）。除此之外，还有一些故障类型，如“接口复杂度”和“设计/代码复杂度”也可以看作既是故障的类型又是故障发生的原因。不过，有相当一部分的“内部功能”类故障未能给出深层的原因，这一点让我们很意外。这类故障一般来说应该是能找到发生原因的。

表25-10列出了各个深层诱因所引发故障的检测难度权重。结果和我们的直觉完全相反：未给出原因的故障是第二难检测出的故障，而“被迫提交”的故障最难被检测出来 。

表25-10　深层诱因的检测难度权重

深层诱因	百　分　比	权　　重	故障描述
8	91/9/0/0	109	错误的修改
7	74/18/7/1	135	认识不足
3	60/40/0/0	140	模棱两可的需求
5	66/27/7/0	141	模棱两可的设计
2	70/17/13/0	143	不完整/遗漏的需求
4	68/25/7/1	143	不完整/遗漏的设计
6	73/12/0/12	147	早前错误的修改
10	76/12/0/12	148	其他
1	63/25/11/1	150	未给出
9	50/46/4/0	158	被迫提交

从对深层诱因的检测难度权重的卡方分析结果来看，由“认识不足”引发的故障常常比预期的更容易找出，而要找出由“被迫提交”引发的故障常常比预期要稍难一些。后一个发现很有意思，因为我们对“被迫提交”的故障几乎一无所知。

在表25-11中，我们计算了深层诱因的修复难度权重。新的排列顺序产生了一些变化：“不完整/遗漏的设计”上升了很多，“模棱两可的需求”和“不完整/遗漏的需求”上升了一些，“未给出”下降了很多，“不清晰的设计”和“其他”下降了一些。不过，各个深层诱因的相对排名并没有什么变化。

表25-11　根据修复难度加权的深层诱因

深层诱因	百　分　比	权　　重	故障描述
10	37/42/12/10	340	其他
1	43/43/12/2	412	未给出
5	29/55/14/2	464	模棱两可的设计
7	30/50/17/3	525	认识不足
6	34/45/17/4	544	早前错误的修改
9	18/57/25/0	564	被迫提交
8	18/55/27/0	588	错误的修改
4	23/50/22/5	653	不完整/遗漏的设计
2	26/44/24/6	698	不完整/遗漏的需求
3	25/30/24/6	940	模棱两可的需求

这个根据修复难度加权的排名正好和人的直觉相符。

对深层诱因的修复难度权重的卡方分析显示，“未给出原因”的故障比预期花费更少的时间来修复，而由“不完整/遗漏的设计”和“被迫提交”造成的故障通常需要超过预期的时间来修复。

在表25-12中，我们用交叉表格来展示各类故障及其深层诱因的数据。横排是各类故障（表格左侧），纵列是各种故障发生的深层原因（用表格上方数字表示）。表中的数字代表着所占总数的百分比。也就是说，1.5%的故障是语言陷阱类故障（即1号故障），由1号原因造成。要计算其预估百分比可以将1号故障百分比和1号原因的百分比相乘，即：20.5% × 3.5% = 0.7%。也就是说，实际观察到的结果比预期要高。

表25-12　故障及深层诱因

		1	2	3	4	5	6	7	8	9	10
		20.5%	**5.4%**	**2.0%**	**25.2%**	**9.8%**	**7.3%**	**17.8%**	**1.1%**	**6.8%**	**4.1%**
1. 语言陷阱	**3.5%**	1.5	0	0	0.2	0.1	0.2	0.8	0.1	0.5	0.1
2. 协议	**0.4%**	0	0	0.1	0.2	0	0	0.1	0	0	0
3. 底层逻辑	**7.9%**	3.7	0.3	0.1	0.6	0.3	1.2	0.7	0	0.6	0.4
4. CMS复杂度	**0.3%**	0.1	0	0	0	0	0.1	0.1	0	0	0
5. 内部功能	**25.0%**	3.3	1.3	0.6	7.7	2.8	2.0	5.2	0.3	1.2	0.6
6. 外部功能	**3.6%**	0.7	0.3	0.1	0.4	0.5	0.6	0.7	0	0.3	0
7. 原始数据类型误用	**2.4%**	0.4	0	0	0.5	0	0.1	0.8	0	0	0.6
8. 原始数据类型不支持	**0.6%**	0	0.2	0	0.1	0	0.1	0.1	0	0.1	0
9. 变更协调	**4.9%**	1.1	0	0	0.8	1.0	0.6	0.8	0.1	0.3	0.2
10. 接口复杂度	**11.4%**	2.1	0.6	0.2	4.1	1.4	1.1	1.4	0.2	0	0.3
11. 设计/代码复杂度	**7.7%**	1.3	0	0.3	3.0	1.6	0.2	1.0	0	0	0.3
12. 错误处理	**3.3%**	0.9	0.3	0	0.8	0	0.1	0.7	0	0.4	0.1
13. 争用情况	**4.3%**	1.4	0.2	0	1.3	0.5	0.1	0.3	0	0.4	0.1
14. 性能	**0.9%**	0.2	0	0.1	0.2	0	0	0.3	0	0	1
15. 资源分配	**1.5%**	0.5	0	0	0.3	0.1	0	0.4	0.1	0	0.1
16. 动态数据设计	**0.3%**	0	0	0	0.1	0	0	0.1	0	0.1	0
17. 动态数据使用	**2.1%**	0.7	0.1	0	0.2	0.1	0	0.6	0	0.4	0
18. 静态数据设计	**1.0%**	0.3	0.1	0.1	0.2	0.1	0	0.1	0	0.1	0
19. 未知的互动	**0.7%**	0	0.1	0.1	0	0.2	0	0.2	0	0.1	0
20. 意外的依赖性	**8.0%**	0.5	0.8	0.3	2.7	0.5	0.1	1.4	0	1.7	0
21. 并行的工作	**4.4%**	0.6	0.3	0	1.2	0.2	0.4	0.9	0.2	0.4	0.2
22. 其他	**5.8%**	1.2	0.8	0	0.6	0.4	0.4	1.1	0.1	0.2	1.0

为简洁起见，我们只列出了最常见的故障和造成这些故障的最主要的一些深层原因。“不完整/被省略的设计”（4号原因）是所有列出故障的最主要成因。“模棱两可的设计”（5号原因），“知识不足”（7号原因）和“未给出”（1号原因）也是造成故障的重要原因。

- 内部功能（故障5）

  统计显示，“不完整/遗漏的设计”（原因5）被认为引发了31%的内部功能类故障，比预计的高1个百分点。“认识不足”（原因7）被认为引发了21%的内部功能类故障。“未给出”是第三多的深层原因，引发了13%的此类故障。

- 接口复杂度（故障10）

  引发此类故障的最主要原因被认为和前一个相同，是“不完整/遗漏的设计”，占36%，比预期要高。“认识不足”和“模棱两可的设计”被认作是第二和第三的主要原因，分别占13%和12%。

- 意外的依赖性（故障20）

  不出意料，“不完整/遗漏的设计”被认为是引发这类故障的主要原因，占34%。“被迫提交”（原因9）引发了21%的此类故障，比预期高1个百分点。“认识不足”是第三大诱因，引发了18%的此类故障。

- 设计/代码复杂度（故障11）

  同上，“不完整/遗漏的设计”被认为是此类故障的主要诱发原因，占39%，比预期高1个百分点。“模棱两可的设计”是第二常见的诱发原因，对21%的此类故障负责，也比预期高1个百分点。“未给出”是第三大诱发原因，占17%。

仍然是为了简洁起见，我们只考虑最常出现的深层诱因及这些原因所引发的故障。

- 不完整/遗漏的设计（原因4）

  我们之前留意到，“内部功能”、“接口复杂度”、“代码/设计复杂性”和“意外的依赖性”是由这个原因造成的主要故障（分别占31%，12%，12%和11%），前三者都比预期的要高。

- 未给出（原因1）

  “底层逻辑”（故障3）是最主要的故障，占这个原因所引发故障总数的18%，比预期高1个百分点。“内部功能”（故障5）是第二大的故障，占16%，比预期低1个百分点。“接口复杂度”（故障10）是第三大故障，占10%。“语言陷阱”是第四大故障，占8%，比预期高1个百分点。

- 认识不足（原因7）

  “内部功能”类故障是最多的故障，占29%，比预期高1个百分点。“接口复杂度”第二，占8%，比预期低1个百分点。“意外的依赖性”第三，占8%。“其他”（故障22）第四，占6%。

- 模棱两可的设计（原因5）

  “内部功能”类故障最多，占29%。“代码/设计复杂性”（故障11）第二，占16%，比预期高1个百分点。“接口复杂度”第三，占14%。“变更协调”第四，占10%，比预期高1个百分点。

5. 预防措施

表25-13按出现频率列出了各个预防措施。我们发现这些结果可能较好地反应了被调查者选择措施的方法。（例子请参见本节稍后关于正式和非正式的预防措施的讨论。）

表25-13 故障的预防措施

预防措施	频率 %	措施描述
5	24.5%	程序演练
6	15.7%	专业人士/文档
8	13.3%	严格遵守准则
2	10.0%	需求/设计模板
9	9.9%	更好的测试安排
1	8.8%	更正规的需求
3	7.2%	更正规的接口规格文档
10	6.9%	其他
4	2.2%	培训
7	1.5%	设计/代码并发性

有趣的是我们注意到特定于该应用程序的预防措施（即“程序演练”）被认为是最有效的措施。这个将程序演练选择为最有用的故障预防措施正好印证了Curtis，Krasner和Iscoe [7] 的观察，即在大型系统的开发中最大的问题是对程序的认知不足。

此外，值得注意的是非正式的措施排在了正式措施的前面。一方面，这也许反映了美国人对于正式措施的一贯偏见。而另一方面，非正式措施代表着非技术类的解决方案。虽然正式的措施也能够用技术手段达到类似的效果，但是可能维护的成本就更高了。

若将这些措施所对应故障按检测工作量来排序，结果和表25-13中的顺序并无甚区别。只有两个例外，一个是“需求/设计模板”，其对应的故障似乎比预计更容易找到；另一个是“严格遵守准则”，其对应的故障似乎比预计更难找到。

在卡方分析的结果中，故障的检测和预防是最为相互独立的关系，$p = 0.041$。“程序演练”对应的故障检测要比预期的稍微容易一些，而“严格执行准则”对应的故障比预期的更难检测。

在表25-14中，我们用故障修复的时间来计算预防措施的权重。

表25-14 预防措施的故障修复时间权重一览

预防措施	百分比（易/中/难/很难）	权 重	措施描述
8	38/52/7/3	389	严格执行准则
9	35/52/12/1	401	更好的测试安排
7	40/40/20/0	460	设计/代码并发性
5	33/50/17/1	468	程序演练
10	49/36/6/9	517	其他
2	10/52/30/1	654	需求/设计模板

（续）

预防措施	百分比（易/中/难/很难）	权 重	措施描述
3	26/43/26/4	675	更正规的接口规格文档
6	22/48/24/6	706	专业人士/文档
1	20/50/22/8	740	更正规的需求
4	23/36/23/18	1016	培训

一个有趣的发现是，被认为通过培训就能预防的故障是最难解决的。而正式的预防措施对应的大都是需要很长时间才能解决的故障。

在用故障修复时间加权之后，预防措施的比例有了一些变化："程序演练"，"更好的测试安排"和"严格执行准则"在比例上都下降了；"专业人士/文档"和"更正规的需求"上升了；"更正式的接口规格文档"和"其他"也上升了少许。结果就是，原来的排序（5，6，2，1，8，10，3，9，4，7）有了这些变化："专业人士/文档"和"更正规的需求"（编号6和11）的排名上升了很多；"需求/设计模板"、"更正规的接口规格文档"、"培训"和"其他"（编号2、3、4和10）上升了少许；"严格执行准则"和"更好的测试安排"（编号8和9）下降了很多。

卡方分析的结果显示，"程序演练"、"严格执行准则"和"其他"措施所能预防的故障通常需要较少天数来解决，而"更正规的需求"、"需求/设计模板"和"专业人士/文档"措施所能预防的故障需要比预期更长的时间来解决。

在表25-15中，我们用交叉表格展示了故障及预防措施的关系。和上次一样，横排的是故障（表格左侧），纵列的是预防措施（表格上方数字）。表格的各项所代表的意思和前一个展示故障及深层诱因的交叉表格类似。

为了简洁起见，我们只考虑最常见的故障及其主要预防措施。"程序演练"被认为是能最有效地预防这些常见故障的措施。此外，"专业人士/文档"，"更正规的需求"和"更正规的接口规格文档"也是预防这些常见故障的重要措施。

- 内部功能（故障5）

  "程序演练"（预防措施5）被视作是预防这个类型故障最有效的措施，可以预防27%的内部功能类故障。"专业人士/文档"（预防措施6）被视作是第二有效的措施，可以预防18%。"需求/设计模板"被视作可以预防14%的此类故障，比预计的稍高1个百分点。

表25-15 故障及预防措施

		1	2	3	4	5	6	7	8	9	10
		8.8%	**10.0%**	**7.2%**	**2.2%**	**24.5%**	**15.7%**	**1.5%**	**13.3%**	**9.9%**	**6.9%**
1. 语言陷阱	**3.5%**	0	0.1	0.1	0	1.0	0.3	0.1	1.3	0.4	0.2
2. 协议	**0.4%**	0.1	0.2	0	0	0.1	0	0	0	0	0
3. 底层逻辑	**7.9%**	0.1	0	0.1	0.2	2.3	0.3	0.2	3.2	0.8	0.7
4. CMS复杂度	**0.3%**	0	0	0	0	0	0.1	0	0.1	0.1	0
5. 内部功能	**25.0%**	1.9	3.5	1.5	0.4	6.6	4.4	0.2	3.3	3.1	0.1
6. 外部功能	**3.6%**	0.6	0.3	0.4	0	0.1	0.7	0	0.5	0.9	0.1

（续）

		1	2	3	4	5	6	7	8	9	10
		8.8%	**10.0%**	**7.2%**	**2.2%**	**24.5%**	**15.7%**	**1.5%**	**13.3%**	**9.9%**	**6.9%**
7. 原始数据类型误用	**2.4%**	0.1	0.1	0.2	0	0.8	0.3	0	0.1	0.2	0.6
8. 原始数据类型不支持	**0.6%**	0.1	0	0	0	0.3	0	0	0	0.1	0.1
9. 变更协调	**4.9%**	0.4	0.9	0.3	0.4	0.8	0.3	0.3	0.3	0.7	0.5
10. 接口复杂度	**11.4%**	2.1	0.3	2.1	0	3.0	1.7	0.1	1.0	0.7	0.2
11. 设计/代码复杂度	**7.7%**	0.8	0.5	0.1	0.4	2.2	2.4	0.2	0.3	0.4	0.4
12. 错误处理	**3.3%**	0.2	0.2	0.3	0.1	0.6	0.6	0	0.4	0.5	0.4
13. 争用情况	**4.3%**	0.8	0	0.4	0	1.2	0.4	0.2	0.4	0.2	0.7
14. 性能	**0.9%**	0	0	0	0.2	0.2	0.3	0	0	0	0.2
15. 资源分配	**1.5%**	0.1	0.1	0.1	0	0.3	0.3	0	0.3	0.3	0
16. 动态数据设计	**0.3%**	0	0	0	0	0.1	0	0	0.1	0	0.1
17. 动态数据使用	**2.1%**	0	0	0.2	0	0.8	0.5	0	0.5	0	0.1
18. 静态数据设计	**1.0%**	0.1	0.1	0	0	0.2	0.2	0	0	0.3	0.1
19. 未知的互动	**0.7%**	0.1	0	0.2	0	0	0.2	0	0	0.2	0
20. 意外的依赖性	**8.0%**	0.6	2.2	1.1	0.1	2.3	0.6	0	0.4	0.6	0.1
21. 并行的工作	**4.4%**	0.4	0.7	0	0.2	1.2	1.1	0.1	0.3	0	0.4
22. 其他	**5.8%**	0.3	0.8	0.1	0.2	0.4	1.0	0.1	0.6	0.4	1.9

- 接口复杂度（故障10）

  同样的，“程序演练”被视作是最有用的预防措施，可以预防26%的该类故障。“更正规的需求”（预防措施1）和“更正规的接口规格文档”被认为是同样有效的措施，都可以预防18%的该类故障，这两者的比率都比预期高一个百分点。

- 意外的依赖性（故障20）

  “程序演练”被视作是最有用的预防措施，可以预防29%的该类故障。“需求/设计模板”被认为是第二有用的措施，可以预防28%的该类故障（比预计的高1个百分点）。“更正式的接口规格文档”被认为可以预防14%的该类故障，比预计的高1个百分点。

- 设计/代码复杂度（故障11）

  “专业人士/文档”被视作是最有效的预防措施，可以预防31%的该类故障（比预期要高）。“程序演练”被认为是第二有效的措施，可以预防29%的该类故障。“更正规的需求”是第三有效的措施，可预防10%的此类故障。

跟之前一样，为了简洁起见，我们只考虑最常见的预防措施以及这些措施可以预防的故障。不出所料，最容易被这些常见预防措施预防的故障是“内部功能”和“接口复杂度”这两类最普遍的故障。不过让人费解的是，最容易被这些预防措施预防的故障还包括“低层逻辑”类故障。

- 程序演练（预防措施5）

  这个措施预防最多的是“内部功能”（故障5），占该措施预防故障总数的27%。“接口复杂度”（故障10）第二，占12%。“低层逻辑”（故障3）和“意外的依赖性”（故障20）第三，各占9%。

- 专业人士/文档（预防措施6）

  和前一条一样，这个措施预防最多的是“内部功能”，占预防总数的29%。“设计/代码复杂度”第二，占15%，比预计的要高1个百分点。“接口复杂度”第三，占11%，比预计要高。

- 严格执行准则（预防措施8）

  这个措施预防最多的两个故障是“内部功能”和“底层逻辑”，分别占预防总数的25%和24%（后者比预期高）。“语言陷阱”（故障1）第三，占10%，比预期要高。“接口复杂度”第四，占预防总数的9%。

6. 深层诱因和预防措施

在表25-16中，可以留意到一个有意思的现象，那就是卡方分析的结果显示了很大的偏差，即对比深层诱因和预防措施的关系数据的预期值和实际值的差别变大了。这意味着故障的深层诱因和其预防措施之间有着强烈的关联。这和我们的常识一致。

表25-16 预防措施及深层诱因

		1	2	3	4	5	6	7	8	9	10
		20.5%	**5.4%**	**2.0%**	**25.2%**	**9.8%**	**7.3%**	**17.8%**	**1.1%**	**6.8%**	**4.1%**
1. 更正规的需求	**8.8%**	0.4	2.3	0.9	3.5	0.8	0.3	0.5	0.1	0	0
2. 需求/设计模板	**10.0%**	0.4	1.7	0.1	3.7	1.9	0.1	0.8	0	1.3	0
3. 更正规的接口规格文档	**7.2%**	0.8	0.3	0.1	2.7	0.8	0.3	2.0	0	0.2	0
4. 培训	**2.2%**	0.4	0	0.1	0.7	0.1	0.3	0.6	0	0	0
5. 程序演练	**24.5%**	7.5	0.2	0.3	7.3	3.1	1.8	3.1	0	0.5	0.7
6. 专业人士/文档	**15.7%**	1.5	0.4	0.4	3.5	1.8	1.0	5.8	0.6	0.3	0.4
7. 设计/代码同步	**1.5%**	0.4	0	0	0.6	0.2	0.1	0.2	0	0	0
8. 严格执行准则	**13.3%**	4.0	0.1	0	0.6	0.2	1.6	2.5	0	3.7	0.6
9. 更好的测试安排	**9.9%**	2.8	0.2	0	1.7	0.8	1.6	1.9	0.3	0.2	0.4
10. 其他	**6.9%**	2.3	0.2	0.1	0.9	0.1	0.2	0.4	0.1	0.6	2.0

我们首先总结一下最主要的深层诱因和它们相对的预防措施。最主要的预防措施分别是“程序演练”，“专业人士/文档”和“严格执行准则”。

- 不完整/遗漏的设计（原因4）

  由此类原因引起的故障有28%被认为可以用“程序演练”（措施5）来预防，比预期高1

个百分点，是最主要的措施。并列排在第二的是“需求/设计模板”（措施2）和“专业人士/文档”（措施6），均被认为可预防14%的此类原因引起的故障，前者比预期高1个百分点。“更正规的需求”（措施1）第三，可预防12%的故障，比预期高1个百分点。

- 未给出（原因1）

  同上，“程序演练”是最主要的措施，被视作可以预防37%的此类故障。接下来的三个措施分别是“严格执行准则”（措施8）、“更好的测试安排”（措施9）和“其他”（措施10），分别被视作可预防19%、14%和10%的此类故障。对于所有的这些措施，实际观察到的值都比预期的要高。

- 认识不足（原因7）

  “专业人士/文档”被认为可以预防32%由这个原因引起的故障，比预期高1个百分点，是对应此项的最主要的预防措施。“程序演练”、“严格执行准则”和“更正规的接口规格文档”分别被认为可以预防17%、14%及11%的此类故障。“程序演练”的比率比预期低一些，而“更正规的接口规格文档”比预期要高一些。

接下来我们反过来，总结一下主要的预防措施，以及他们对应的故障诱因。这些预防措施预防最多的诱因分别是“认识不足”、“未给出”和“模棱两可的设计”。有一点令人不解的是，“未给出”竟然被视作这些预防措施能预防的主要诱因之一。

- 程序演练（措施5）

  “未给出”（原因1）和“不完整/遗漏的设计”（原因4）被认为是这个措施主要预防的诱因，分别占总数的31%和30%，比预期要高。“模棱两可的设计”（原因5）和“认识不足”（原因7）并列第二，分别占13%，前者比预期高，后者比预期低。

- 专业人士/文档（措施6）

  “认识不足”被看做是这个措施预防得最多的故障诱因，占总数的37%，比预期要高1个百分点。“不完整/遗漏的设计”和“模棱两可的设计”被认为是第二和第三，分别占23%和11%。“未给出”是第四，占10%，比预期要低。

- 严格执行准则（措施8）

  “未给出”和“错误的修改”被认为是这个措施预防最多的两个故障诱因，分别占30%和28%，均高出预期水平。“认识不足”和“早前修复不当”占第三和第四，分别占19%和12%，后者比预期要高。

25.4.3　界面故障与实现故障

对于什么是界面故障，我们这里采取 Basili 和 Perricone[2]以及Perry 和Evangelist [19][20] 的定义，即：界面故障是指“处在模块之外，但又被模块所使用的组织结构的相关故障”。从这个定义来看，我们大概可以将“语言陷阱”（1），“底层逻辑”（3），“内部功能性”（5），“设计/代码复杂性”（11），“性能”（14）和“其他”（22）作为实现故障，而剩余的则是界面故障。刚刚说这样的分类是“大概”的原因是，有一些实现类的故障有可能也包含着界面的问题。例如之前我

们看到有些“设计/代码复杂性”的故障被认为可以由“更正规的接口规格文档”来解决。表25-17列出了两种故障的对比。

表25-17　接口/实现故障对比

	接口故障	实现故障
频率	49%	51%
用检测难度加权后	50%	50%
用修复难度加权后	56%	44%

虽然比起实现故障，接口故障的频率要小一些，但是两种故障却需要同样的时间来检测，而且接口故障比实现故障需要更多的时间来修复。

表25-18对两种故障以及各自的深层诱因进行了对比。“其他”、“模棱两可的需求”、“未给出”、“早前修复不当”和“模棱两可的设计”更偏向于造成更多的实现故障。而“不完整的/遗漏的需求”、“错误的修改”和“被迫提交”较为倾向于造成更多接口故障。

需要注意的是，从表中的数据来看，那些和模棱两可、暧昧不清相关的诱因通常引发更多的实现故障，而那些和不完整性、遗漏和疏忽相关的诱因通常引发更多的接口故障。

表25-18　接口/实现故障和深层诱因

		接口故障	实现故障
		49%	**51%**
1	未给出	45.2%	54.8%
2	不完整/遗漏的需求	79.6%	20.4%
3	模棱两可的需求	44.5%	55.5%
4	不完整/遗漏的设计	50.8%	49.2%
5	模棱两可的设计	47.0%	53.0%
6	早前的修复不当	45.1%	54.9%
7	认识不足	49.2%	50.8%
8	错误的修改	54.5%	45.5%
9	被迫提交	63.1%	36.9%
10	其他	39.1%	60.1%

表25-19对比了接口和实现故障及对应的预防措施。不出意料，比起预防实现故障，措施1和措施3能预防更多的接口故障。措施8，措施4和措施6被认为更能预防实现类的故障。

表25-19　接口/实现故障和预防措施

		接口故障	实现故障
		49%	**51%**
1	更正规的需求	64.8%	35.2%

（续）

		接口故障	实现故障
		49%	**51%**
2	需求/设计模板	51.5%	48.5%
3	更正规的接口规格文档	73.6%	26.4%
4	培训	36.4%	63.6%
5	程序演练	48.0%	52.0%
6	专业人士/文档	44.3%	55.7%
7	设计/代码同步	46.7%	53.3%
8	严格执行准则	33.1%	66.9%
9	更好的测试计划	48.0%	52.0%
10	其他	49.3%	50.7%

25.5　研究结果可靠吗

做实证研究就和做软件系统一样，我们不可能创造出一个完美的系统，而且常常都会犯错误，做出存在各式各样问题和缺点的系统。不过无论是做研究还是做软件，最重要的关键都是同一个，那就是：这些缺点和瑕疵是否会影响到研究或者软件系统的有效性？

要确定我们研究的优劣以及研究结果是否有效，我们需要回答下面三个问题：（1）我们调查的对象是否正确；（2）对于我们观察到的结果，有没有别的解释（即我们的方法是否正确）；（3）我们的研究结果的意义何在（即我们能用这些结果来做什么）。

25.5.1　我们调查的对象是否正确

我们有非常充分的理由相信我们的研究能帮助人们理解软件开发中出现的故障及其带来的后果。在研究中我们解答了这样一些基本的问题：出现的故障，其检测的难度，修复的难度，产生的深层原因以及如何来预防、检测、改善这些故障。此外，我们还回答了早前在接口故障研究中提出的一个疑问：接口故障和实现故障，哪个更难检测/修复？卡方分析显示故障及其深层诱因和预防措施之间有着很强的关联性，而前后一致并相互呼应的分析结果成为了我们的答案的强力支持。这种强烈的关联性也意味着我们的各个调查结果是一致的。

但是，这份研究仍然存在着缺陷。首先，故障的分类并不仔细。其次，故障检测和故障修复的难度测量方法并不统一，故障修复难度的测量方法要比检测难度的方法好得多。最后，没有清楚地区分接口故障和实现故障。

这样的故障分类的主要优点是：它是由开发人员（而非研究人员）自己定义的。缺点就是分类不仔细而且太长了。在实际参与调查的过程中，被调查者很有可能没有耐心看完整个列表来选择最符合的类别，而是匆匆选择比较接近的靠前的类别。

在后续进行的研究[13] [14]中，我们解决了故障分类的问题。方法是将故障分为三大类，分别

是实现、接口和外部故障（这样也同时解决了上面说的第三条缺陷），而在这些大类之下又有6到8个小类（见参考文献[14]的117页）。

同样的，检测和修复故障的难度测量方法也是由开发人员决定，但是后果就不是分类不仔细那么简单了。测量修复故障的难度很简单，只需要记住到底这个故障花了多少时间即可，如1天，1周或者1个月，不容易搞错。而对于检测故障的难度，就不同了。由于涉及很多主观的因素以及人与人之间的差异（基于管理上的限制我们无法确定这种差异性），检测故障的难度很难量化。我们推荐的做法是用统一的可以量化的指标（例如时间）来衡量检测和修复的难度。当然我们在后续的研究[13] [14]中正是这样做的。

接口和实现故障我们区分得并不是特别清楚，有的接口故障中包含了一些实现的成分，反之亦然，所以在这份研究中我们也仅仅是大概区分了一下。如前面所言，我们在后续研究[13] [14]中分了三大类（实现、接口、外部）并以此解决了这个问题。

25.5.2 我们的方法是否正确

在我们进行的两个步骤中，共收回了68%的问卷——也即是说，我们的数据涉及了大概2/3的故障。鉴于进行调查的环境比较苛刻，这样的回复率已经大大超出了我们的预期。诚然，数据量也是数据是否可靠的一个重要指标。

虽然我们没能给出最详尽的数据，但是我们已经尽力在对于调查结果的讨论中做到精确。为了理解方便，我们只精确到了小数点后一位数，事实上，我们所掌握的数据量已经大到可以精确到小数点后两位数。

和所有的调查研究一样，我们很难说那些没有收回的问卷和没有参与调查的人能在多大程度上影响我们的调查结果。幸运的是，我们了解到，在调查过程中，并没有出现已知的会影响调查结果的因素，如只反馈简单的或者困难的故障，或者只有初级或者高级的程序员参与调查等[1]。

我们还提到了关于调查的重要条件限制：第一，调查绝对不能打扰正常工作；第二，调查必须是完全自愿参加的；第三，调查必须是完全匿名的。由于这些管理层的要求，我们无法验证调查结果的真实性[3]和准确性。不过，有两个事情可以减轻无法验证调查结果所带来的影响：第一，问卷是由作者和一组开发人员一起设计的；第二，问卷经过了另一组开发人员的独立评估。在调查后做验证是为了确保被调查者对于问卷问题的正确解读，但我们相信，我们在调查前所做的工作也能达到同样的效果，因为：第一，我们让开发人员参与到问卷的制作中来，确保问卷是用他们熟知的术语写成的；第二，我们在正式开始调查之前对一小组开发人员进行了测试调查，而在测试调查中，我们没有发现任何错误解读问卷的情况。

剩下的问题，就是时间间隔过长了。从故障解决到填写问卷，中间的时间最长达到了整整一年。在这个过程中，相关的信息有可能会有所丢失和遗忘。不过，让主导解决这个故障的人做调查，还是比完全不了解这个故障的人好得多。要想解决时间推移的问题就必须“乘热打铁”，把故障调查作为解决故障的标准流程的一部分。

还有一个需要注意的是：我们只取了测试阶段的故障数据，这可能对总体调查结果有所影响。

如果我们能收集到从需求分析阶段到系统发布时的所有故障，就能得到一个更为全面和准确的结果。但是，我们也注意到，这种从测试阶段开始才记录故障信息的方式对于大部分软件开发流程来说几乎可以说是一种标准，所以这个问题可以忽略。

总的来说，我们相信这些调查研究设计上的问题，相比起支持我们论点的证据而言，是微不足道的。这也意味着，我们的数据是有效的，调查结果并未受其他因素的影响。

25.5.3　我们能用这些结果做什么

所有软件行业的实证研究都需要回答的一个主要问题是："对于我这样的一个开发人员来说，这些结果意味着什么？"这个问题的答案部分取决于研究是否具有足够的代表性，这个问题可以从两个角度来回答。

第一个角度是，到底我们所研究的这个发行版本在多大程度上可以代表这个系统所有发行版本的情况。如果这个发行版本不具备代表性，那么基于这个发行版本的研究结果也不一定具备代表性。在这个研究中，我们认为所选定的发行版本具备代表性，因为结果中故障修复、新功能和改良的比例和之前的版本一致。在我们调查这个发行版本之后的几个版本中，故障修复的比例有所上升，不过，在这几个版本之后，整个比例又恢复到了之前的状态。

在确定版本代表性的问题之后，我们需要确定所研究的软件系统能在多大程度上代表其他的软件系统。对于这个问题，可以确定的是，我们选择的这个系统可以代表具有与其同样特征，如大规模、高度容错、极高稳定度、实时性，并在Unix环境下使用常用编程语言（如C）来开发的这一小类系统。在这样的系统中存在着类似的问题。

那么这个系统的开发和其他软件系统的开发有多少是相通的呢？我们认为非常相通。在最主要的五种故障中，我们并未发现任何证据表明这些主要的故障仅存在于这一类系统中。实际上，整个故障的分布情况和我们在各种规模或者类型的软件开发中所遇到的大同小异。我们对预防措施也有相同的结论。而主要的区别在于不同的系统会有不同的故障的频率、诱因和预防措施。

我们采用的研究方式毫无疑问是具有普遍性的，适用于对各种规模和类型软件开发的研究。如果在其他系统中也能发现同样常见的故障，那就意味着我们的研究结果也适用于这些系统。对于这一点，我们数据前后的一致性和数据间的相互呼应可以作为证明。

25.6　我们明白了什么

研究的结论总结如下。

- 需求、设计和代码编写相关的故障占总数的34%。需求故障大概占5%。这些故障虽然数量并不是特别大，但却非常重要，因为其发现的时间点通常非常靠后，在发现了以后，修复故障的成本已经非常高了。
- 测试大规模的、复杂的实时系统的实验环境本身又是另外一个大规模的复杂的实时系统。在我们研究的这个发行版本的开发中，测试相关的故障占了总数的四分之一。
- 故障中有16%是"非故障"，再加上很多"未知的依赖性"和"接口或设计/代码复杂度"

这样的故障，都表明了对系统的认识不足是这个版本开发过程中的一个大问题。

- 设计和代码编写相关的故障中，有78%需要5天或者更少的时间解决，有22%需要6天或者更多的时间来解决。我们注意到在修复这些故障时涉及一些额外开销，包括如何达成一致的解决方案、创建需要的系统组件以及用测试环境来验证修复是否正确等。不过可惜的是，我们并没有关于这些额外开销的数据。
- 数量最多的前5个故障类型占到了总数的60%。它们分别是：内部功能、接口复杂度、未知的依赖性、底层逻辑和设计/代码复杂度。除了“底层逻辑”以外，其余几个都是我们预感在这样一个大型而复杂的实时系统中频繁出现的问题。
- 为故障加上了检测和修复时间权重之后，结果正好和我们对于各个故障的检测和修复难度的感觉一致。
- 设计和代码相关的故障的深层诱因中，“不完整/遗漏的设计”，“认识不足”和“未给出”（我们认为“未给出”意味着有时候并没有更深层的诱发原因）占到了64%。这些诱因的故障修复权重和我们的感觉正好一致：由需求引发的故障最难修复，而由模棱两可的设计和认识不足引发的故障最容易修复。
- “程序演练”，“专业人士/文档”，“严格执行准则”和“需求/设计模板”被认为可以预防64%的设计和代码相关故障。我们认为，“程序演练”高达25%的比例可以证明Curtis，Krasner和Iscoe的看法[7]，即对于程序的认识不足是一个重要的问题。
- 虽然人们更倾向于非正式的预防措施，但是非正式的预防措施所预防的故障都常常比较容易解决，而正式的预防措施所预防的故障一般更难解决。
- 在Perry和Evangelist的两篇论文[19][20]中，接口故障被认为占了全部故障的大部分（68%）。但是，研究中接口故障和实现故障并没有加权比较。在我们的研究中发现，接口故障大概占总数的49%，但他们要比实现故障要更难解决（参见前面的讨论）。不出所料，“更正规的需求”和“更正规的接口规格文档”被看作两个重要的预防接口故障的措施。

由于所研究的这个系统的开发使用了当前的“最佳实践”所推崇的技巧和工具，开发人员也都非常称职，所以，我们觉得这里得出的数据也可以适用于其他的大型实时系统。有鉴于此，我们也在这里对改进当前的“最佳实践”提出如下一些建议。

- 在所有开发环节中收集故障数据（而不是只在测试的环节中收集），并用这些数据监测整个开发进度。
- 将故障调查作为故障修复的一个必需步骤来进行，乘热打铁地在开发人员尚未忘记故障信息的时候进行调查。调查的数据会提供一个直观的方式来改进流程，以解决当前最频繁或者最难检测/解决的故障。
- 在当前的流程中加入非正式的、需要大家参与的预防措施，如“程序演练”、“专业人士/文档”和“严格执行准则”等。我们的研究已经表明，这些措施对大部分故障的检测和解决都有好处。
- 在当前的流程中加入正式的工具和方法，以提高需求、架构和设计文档的准确度和完整度。这些可以对那些较难修复的故障产生作用，也可以帮助我们在开发中更早发现需求、

架构和设计上问题。

作为结尾，这里列出我们从研究中所学到经验和教训，以期能帮助改进以后的软件开发流程。

- 如果希望用减少故障的方式来改进产品的话，最快的办法就是雇一群真正了解这个产品的人。记住，认识不足常常是故障的最主要的深层诱因，所以最快的办法就是雇佣认识充分的人。
- 改进软件开发的方法中，最不靠谱的就是使用“更好”的编程语言。我们的研究显示，可以用使用更好的编程语言解决的故障相对较少。
- 在改善开发环境的过程中，我们应该尝试更多地使用各种工具和方法来帮助人们了解系统（及修改系统会带来的后果）。因为说到底，这些提升对系统认识的方法很可能是故障预防的最有效措施。

25.7　致谢

在此，我要对Carol Steig表达特别的谢意，感谢她在这个研究早期和我的合作。感谢David Rosik对故障调查问卷做出的巨大贡献。Steve Brunn做了交叉表格的统计分析，并和Carolyn Larson、Julie Federico、H.C. Wei以及Tony Lenard一起帮助我们分析了调查问卷，感谢大家。感谢Clive Loader让我们加深了对卡方分析的认识。我们尤其要感谢Marjory P. Yuhas和Lew G. Anderson对我们工作孜孜不倦的支持。最后，我们还要感谢所有参与了问卷调查的人们。

25.8　参考文献

[1] [Basili and Hutchens 1983] Basili, Victor R., and David H. Hutchens. 1983. An Empirical Study of a Syntactic Complexity Family. *IEEE Transactions on Software Engineering* SE-9(6): 664-672.

[2] [Basili and Perricone 1984] Basili, Victor R., and Barry T. Perricone. 1984. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM* 27(1): 42-52.

[3] [Basili and Weiss 1984] Basili, Victor R., and David M. Weiss. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* SE-10(6): 728-738.

[4] [Boehm 1981] Boehm, Barry W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

[5] [Bowen 1980] Bowen, John B. 1980. Standard Error Classification to Support Software Reliability Assessment. *Proceedings of the AFIPS Joint Computer Conferences, 1980 National Computer Conference*: 697-705.

[6] [Brooks 1995] Brooks, Frederick P., Jr. 1995. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition. Addison-Wesley.

[7] [Curtis et al. 1988] Curtis, Bill, Herb Krasner, and Neil Iscoe. 1988. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM* 31(11): 1268-1287.

[8] [Endres 1975] Endres, Albert. 1975. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering* SE-1(2): 140-149.

[9] [Fenton and Ohlsson 2000] Fenton, N.E., and N. Ohlsson. 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering* SE-26(8): 797-814.

[10] [Glass 1981] Glass, Robert L. 1981. Persistent Software Errors. *IEEE Transactions on Software Engineering* SE-7(2): 162-168.

[11] [Graves et al. 2000] Graves, T.L., A.F. Karr, J.S. Marron, and H. Siy. 2000. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering* SE-26(7): 653-661.

[12] [Lehman and Belady 1985] Lehman, M.M., and L.A. Belady. 1985. *Program Evolution: Processes of Software Change*. London: Academic Press.

[13] [Leszak et al. 2000] Leszak, Marek, Dewayne E. Perry, and Dieter Stoll. 2000. A Case Study in Root Cause Defect Analysis. *Proceedings of the 22nd International Conference on Software Engineering*: 428-437.

[14] [Leszak et al. 2002] Leszak, Marek, Dewayne E. Perry, and Dieter Stoll. 2002. Classification and Evaluation of Defects in a Project Retrospective. *Journal of Systems and Software* 61(3): 173-187.

[15] [Musa et al. 1987] Musa, J.D., A. Jannino, and K. Okumoto. 1987. *Software Reliability*. New York: McGraw-Hill.

[16] [Ostrand and Weyuker 1984] Ostrand, Thomas J., and Elaine J. Weyuker. 1984. Collecting and Categorizing Software Error Data in an Industrial Environment. *The Journal of Systems and Software*, 4(4): 289-300.

[17] [Ostrand and Weyuker 2002] Ostrand, Thomas J., and Elaine J. Weyuker. 2002. The Distribution of Faults in a Large Industrial Software System. *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*: 55-64.

[18] [Ostrand et al. 2005] Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2005. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering* SE-31(4): 340-355.

[19] [Perry and Evangelist 1985] Perry, Dewayne E., and W. Michael Evangelist. 1985. An Empirical Study of Software Interface Errors. *Proceedings of the International Symposium on New Directions in Computing*: 32-38.

[20] [Perry and Evangelist 1987] Perry, Dewayne E., and W. Michael Evangelist. 1987. An Empirical Study of Software Interface Faults—An Update. *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences* II: 113-126.

[21] [Perry and Steig 1993] Perry, Dewayne E., and Carol S. Steig. 1993. Software Faults in Evolving a Large, Real-Time System: A Case Study. In *Lecture Notes in Computer Science, Volume 717: Proceedings of the 4th European Software Engineering Conference*, ed. I. Sommerville and M. Paul, 48-67.

[22] [Perry and Wolf 1992] Perry, Dewayne E., and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17(4): 40-52.

[23] [Perry et al. 2001] Perry, Dewayne E., Harvey P. Siy, and Lawrence G. Votta. 2001. Parallel Changes in Large Scale Software Development: An Observational Case Study. *Transactions on Software Engineering and Methodology* 10(3): 308-337.

[24] [Purushothaman and Perry 2005] Purushothaman, Ranjith, and Dewayne E. Perry. 2005. Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Transactions on Software Engineering* SE-31(6): 511-526.

[25] [Rowland et al. 1983] Rowland, B.R., R.E. Anderson, and P. S. McCabe. 1983. The 3B20D Processor & DMERT Operating System: Software Development System. *The Bell System Technical Journal* 62(1/2): 275-290.

[26] [Schneidewind and Hoffmann 1979] Schneidewind, N.F., and Heinz-Michael Hoffmann. 1979. An Experiment in Software Error Data Collection and Analysis. *IEEE Transactions on Software Engineering* SE-5(3): 276-286.

[27] [Shao et al. 2007] Shao, D., S. Khurshid, and D. Perry. 2007. Evaluation of Semantic Interference Detection in Parallel Changes: An Exploratory Experiment. *Proceedings of the 23rd IEEE International Conference on Software Maintenance*: 74-83.

[28] [Siegel et al. 1988] Siegel, Sidney, and N. John Castellan, Jr. 1988. *Nonparametric Statistics for the Behavioral Sciences*, Second Edition. New York: McGraw-Hill.

[29] [Thayer et al. 1978] Thayer, Thomas A., Myron Lipow, and Eldred C. Nelson. 1978. Software Reliability—A Study of Large Project Reality. In *TRW Series of Software Technology*, Volume 2. Amsterdam: North-Holland.

[30] [Thione and Perry 2005] Thione, G. Lorenzo, and Dewayne E. Perry. 2005. Parallel Changes: Detecting Semantic Interferences. *Proceedings of the 29th Annual International Computer Software and Applications Conference*: 47-56.

第26章

新手专家：软件行业的应届毕业生们

Andrew Begel
Beth Simon

关于软件工程的教育已经有很多论述了：如何教初学计算机的人们学会编程、设计和测试等技能，好让他们成为专业的软件工程师。但是，学习不会止步于毕业。实际上，毕业仅仅是新的学习的开始。新进公司的工程师们必须学习在截止时间之前完成代码的编辑、修改和调试，也需要学习用恰当的方式与大型团队里的其他同事们交流和互动。在这一章中，我们将详细展示初入职场的软件工程师的经历，以此来讨论大学和公司两种学习过程的共性以及区别。

大学教育尝试教给学生们关于电脑技术的一些核心概念，这样他们便可以在以后的生活中自发地学习，并跟上计算机科学的潮流。新的编程范式（比如面向对象的编程）和新的开发方法论（比如敏捷、极限编程等）的诞生让软件行业发展迅速，也使得学校教育必须主要关注这些“硬”技能。学校的学术环境并不会教给学生太多“软”技能，也就是软件工程中和“人”有关的部分，比如创建和调试规格文档，为代码加上文档以阐述其原理及历史，遵循软件开发方法论的原则，管理大型项目以及和团队中的其他成员合作等。

那些初入职场的毕业生们需要再学习新的技术、技巧和流程，实际上又变回了一个初学者。他们也许会惊讶的发现，原来在他们刚刚得到的工作中，软技能才是最主要的组成部分[2][9]。若要成为有用之才，尤其是在大型独立的软件开发公司中的软件开发人才，需要完整的软硬技能。而雇主们发现，刚刚大学毕业的应届生们通常不具备这样完整的技能。在eWeek.com上刊登的一篇文章中，多个接受采访的业内软件开发人员都指出，应届毕业生们缺乏沟通和团队合作能力，而且对于复杂的开发流程、旧的代码、截止期限以及用有限资源开发这些工作中常见的东西毫无准备[12]。

Schein认为，新人融入公司的过程包含三个方面：职务、层级和社会网络[10]。

- 职务

  “职务”代表一个职位的任务以及技术要求。大学里的课程教学生们学习通用的知识（比如编程、数据结构、软件工程）和专业领域知识（比如图形设计、人工智能、操作系统），

这些课程可以很好地解决这个问题。

- 层级

  “层级”是公司从上至下的指挥体系。学校并没有很多涉及这个方面。举例来说，虽然在很多的课程中学生们都需要组成小组并进行合作，但是小组中每个人的权力都相同，而且往往经验也相同，这与职场新人需要面对的状况非常不同。

- 社会网络

  “社会网络”代表着职场新人逐渐创造新的人脉关系，并从社会网络的边缘向中心移动的过程。很不幸，在这一点上学校帮不上什么忙，如何发展完全靠学生们自己，而学校提供的忠告就只剩下不要作弊或者不准相互帮忙做家庭作业了。

由于学校教学的局限性，我们认为很多学生在毕业时并没有做好进入软件开发行业的准备。这个结论来自于对在微软工作的8位高校应届毕业生所进行的研究。我们观察了这些新开发人员在进入公司头半年的两个月中日常工作的情况。在分析了这些新晋软件开发人员的任务、活动、社交和产出的数据之后，我们发现，虽然新的开发人员们在职责和技术上并没有问题，但是对于日常的社交和沟通，他们就显得缺乏准备和培训了。我们认为，他们所缺乏的这些东西加重了自身的压力和焦虑感，并影响了他们在开始的几个月中的表现。

对于职场新人的社会化研究告诉我们，要想提高自己的创造力、工作效率和对工作的满意度，掌握职务、层级和社会网络非常重要。如果对新人的培训采取新的教育方法，比如结对编程、合理的边际参与（legitimate peripheral participation）、辅导计划等，那么新人们就可以更容易地掌握这三个方面。我们希望这一章中对于软件开发新手的详细研究可以帮助人们更好地讨论和评估本科教育的各种改良方案。我们鼓励读者在阅读研究数据的时候，也考虑一下是否可以尝试新的教育方案和方法。

26.1　研究方法

我们进行了一次直接观察式的研究并用扎根理论[①]分析了得到的数据。直接观察意味着有一个研究人员将和被观察者坐在同一个办公室，留意被观察者的一举一动，并记录下来。比起问卷调查，我们通常更倾向于直接观察。因为作为客观的观察者，我们可以做到不偏不倚，并可以记录下每一个细节，这样就能够知道发生的各种事件的准确信息。在这一点上，直接观察和采访以及问卷调查正好相反，在后两种调查方法中，参与调查的人很容易受到各种偏差的干扰。最典型的偏差来源于对细节的忽略，最后在调查中就只是给出了一些笼统的日常工作，而不是详细地去描述每一个具体的事情。此外还有记忆偏差，事情一旦久远，人的记忆就会产生偏差。在我们这种定性研究中，最终目标并不是消除所有的偏差（这也不可能做到），而是尽量避免会对我们的调查结果造成很大影响的偏差。

① grounded theory，指和传统的“先做假设再证明”不同的，直接用数据来得出结论的研究方法，“扎根”于数据的研究理论。——译者注

我们的案例研究选择使用少量研究对象进行直接观察。研究对象的选择尽量做到各有不同，这样，我们便可以观察到更多不同的活动、行为和现象。直接观察是一种十分消耗精力的收集数据方法，调查人员必须每天花很长时间跟随调查对象。这也限制了能研究的对象的数量，所以我们选择少量的调查对象，并尽量多地观察他们身上出现的情况。

分析数据使用的方法是扎根理论[11]。扎根理论是一种定性研究的方法，即研究者们只从自己得到的数据中生成结论，而不是先假设一个结论然后去证明它。当我们并不确定将得到什么样的数据，也不知道哪些数据将会是最重要的时候，这种方法就比较实用。扎根理论采用三个步骤来分析数据。

(1) *开放式编号*

记录中的每条数据都被加上了一个或多个标签，用来对其描述。

(2) *主轴编号*

将开放式编号的各个标签相互关联，用来展示因果关系、调节措施以及由于一个行为所导致的另一个行为。

(3) *主题编号*

所有的标签和他们之间的联系被串成一个完整的故事线，而这条故事线通常就是最终结论的基础。

在这一节中，我们将详细介绍我们的观察和分析方法，并展示一些我们所得到的原始数据。

26.1.1 研究对象

我们选择了微软新招的8位开发人员。在调查开始的时候，他们在微软工作的时间在1~7个月之间。我们确定了25个可能的研究对象（基于经理的批准和时间安排上的考虑），并在考虑了学历和职务分步的情况下，最终选定了7位男士和1位女士，共8位开发人员进行研究。在这些开发人员中，4位有学士学位，1位有硕士学位，3位有博士学位，并且所有人的专业都是计算机科学或者软件工程学。其中，有2位是在美国获得的本科学位，2位在中国，1位在墨西哥，1位在巴基斯坦，1位在科威特，1位在澳大利亚。3位博士的博士学位均是在美国大学获得。我们还尽量选择了软件开发经验最少的开发人员（即除了有限的实习之外没有任何工作经验），除了一位来自澳大利亚的开发人员，他在微软之前有两年的开发经验。参与者们每周将得到50美元作为参与研究的报酬。

每一轮观察的时间是两个星期。在两个星期里，我们每天观察这些开发人员6~11个小时，一轮完毕之后间隔一个月，然后继续下一轮观察。我们在不打扰调查对象工作的情况下对他们进行观察，跟随他们到各种会议，并观察他们如何与周围的同事和朋友进行交往。我们将所观察到的事情连同发生时间记录下来，形成将要分析的数据。和电视上常见的真人秀类似，开发人员们还被要求在每个非观察日结束的时候录制一段视频日记。我们要求他们在视频中谈一谈当天最有趣的事情，然后对他们的大学教育或者现在工作的某些方面进行回顾。

26.1.2 任务分析

在扎根理论的开放式编号阶段，我们通过审视观察记录找出这些新手们所做的事情，然后给这些事情加上一到两个任务或者子任务类型。表26-1展示了所使用的任务分类。举例来说，如果新手问同事某个API在哪里定义的，那么这条记录的分类就将是“编程/搜索”和“沟通/问问题”。

表26-1 新手执行的任务，按频率排序

编 程	读、写、加注释、校对、代码审阅
解决bug	重现、报告、排优先级、调试
测试	写、运行
项目管理	提交代码、签出代码、复原代码
文档	读、写、搜索
规范	读、写
工具	套索、寻找、安装、使用、构建
沟通	问问题、说服他人、和他人协调、email、参加会议、准备会议、找人、和领导交流、教别人、向别人学习、做导师

26.1.3 任务案例

在这一节，我们将描述一下参与者被观察的情况，以及相关的任务记录（表26-2所示）。Timothy被指派去修复一个bug。在看文档学习了重现bug的五个步骤之后，他试着照做，但是第一个步骤就失败了。在45分钟内，他尝试用各种方法来进行调试，包括交换软件库甚至交换电脑，但是还是没能成功。他走到另一个办公室去寻求同事Abdul的帮助。Timothy说明了自己所试过的步骤，但是Abdul说他弄错了。然后Abdul和他一起回到了他的办公室，然后发现Timothy原来是将错误的软件库复制到了他的电脑。在告诉他正确的库在哪儿之后Abdul返回了自己的办公室。

不过这个错误软件库的问题是由Timothy的调试方法所引起的，并不是真正的问题所在。Timothy又用了12分钟来重现这个bug，然后又找到Abdul，告诉他还是有问题。Abdul更详细地解释了Timothy复制的那些程序库，让Timothy意识到他对于这些程序库和其应该放置的目录的理解是错误的。Abdul告诉了Timothy正确的理解，并回想了当年他如何英勇解决调试故障的故事，然后告诉了Timothy另一些常用的调试方法，就又让Timothy回去了（Abdul所说的方法Timothy都已经试过了）。Timothy没有继续请Abdul帮忙，而是想去尝试所有Abdul教他（而他认为没用）的正确理解下的方法。然后他又花了25分钟自己进行调试和测试，但是还是不能完成第一个重现bug的步骤，最后什么也没做成。

表26-2 Timothy的活动记录，按任务和子任务分类

时 间	描 述	任务/子任务类型
11:45:43 AM	重新运行复制脚本	解决bug/重现

（续）

时　间	描　述	任务/子任务类型
11:46:18 AM	脚本运行完成。检查输出并确保其正确。脚本给了警告说文件没有签名。关于源目录的邮件又说文件是签了名的。奇怪。复制成功了，但是生成的二进制文件没有签名	解决bug/重现
11:47:26 AM	摇头。Timothy很困惑。组长说这些文件是签了名的。但是事实证明没签	解决bug/重现
11:48:11 AM	Timothy说可能他应该自己为这些文件签名	
11:48:36 AM	Timothy自言自语："坏了坏了"	
11:56:23 AM	Timothy穿过走廊去问Abdul这是什么问题	沟通/问问题
11:56:35 AM	Timothy解释了当前状况，Abdul并不同意，然后两人一起到了Timothy的办公室，最后Abdul说Timothy复制错文件了，和签名不签名没关系。他现在复制了正确的文件（仍然是未签名的），不需要重启	沟通/向他人学习
11:57:27 AM	程序可以运行了	解决bug/重现
11:57:49 AM	再次尝试重现这个bug。URL可以用了。重现失败。Timothy表现出了迷惑——我怎么才能重现这个bug呢？调试和非调试版本的程序都不能重现	解决bug/重现

26.1.4　做回顾的方法

在非观察日，这些参与者们都将录制一份3~5分钟的视频日记。我们准备了40个问题，大部分参与者都选择了在视频中回答前20个问题，有一个人回答了35个。数量的不同来源于缺席，没有时间和我们所做观察的数量。我们观看了视频并记下了13个和学习以及大学经历有关问题的答案。

26.1.5　有效性问题

在我们所做的这种基于个人经验的定性研究中，常常会参杂入现实生活中的各种复杂因素。在调查样本如此小，而且每个人的经历又都各有不同的情况下，我们自然很难100%确定到底是什么造成了我们观察到的结果。而我们的研究方法以及各个参与者都有可能造成结果的偏差。

我们的研究对象来自于不同的教育背景，而且在研究开始的时候分别处在个人和职业发展的不同阶段。此外，在我们进行研究的两个月里，每个人的改变和所学到的东西又各有不同。虽然在这种情况下我们很难对数据做横向比较，不过这没有关系，因为我们的研究主要关注的正是新手的这种学习效应。

在我们的研究中不可避免的一种偏差是*霍桑效应*（Hawthrone effect），即观察者的存在会改变被观察者的行为。我们相信我们连续而长时间的观察将使各位参与者慢慢适应。在研究结束的时候，我们询问各位参与者是否留意到自己在行为上有任何变化，但是大多数人只是说当我们在周围的时候他们的休息少了一些。为了避免公司的层级结构带来的偏差，我们清楚地向新人及他们的经理们说明，所有数据都将是隐私的，我们也不会对任何人公开这个数据，包括公司的管理层。此外，我们也采取了各种措施来确保我们的参与者们不是被自己的经理强迫参与研究的。在任何时候，包括在研究完成以后，参与者都可以退出研究并删除和自己相关的数据。不过并没有人这样做。

在阅读我们的研究结果的时候，有一点必须记住，开发人员们认为自己需要的信息，并不能完全代表做好开发真正需要的信息。尤其是由于采用了不打扰的旁观式研究方式，使得我们发现很多新手有时候并没有意识到他们其实需要某些信息。也就是说，他们可能像没头苍蝇似地乱转，停止前进的步伐，然后就换其他的任务做或者做一些不相关的事情。而在观察者眼中，如果他们能了解到某些信息，就能够有所进展。特别是，我们观察到很多时候新手们并没有意识到他们应该从同事那儿寻找信息，来帮助自己完成任务。我们的数据显示，新手们常常做那些可能不应该他们做的事情。

26.2　软件开发任务

在这一节中，我们将展示一些新手遇到的软件开发任务的技术细节，统计他们在这些细节上所花时间的分布情况，并展示一些他们在完成任务时常做的一些事情。那些最消耗时间的事务正好是一个不错的分析对象，可以让我们确定我们的大学计算机科学教育有没有涉及到这些较难的事务。

图26-1展示了我们的新手所花在各项任务上的时间（已规范化并显示为百分比）。各个项目的柱状图之间可以做横向或者纵向比较，例如，根据柱状图的长度，我们可以看到Timothy花时间最多的事务是沟通。

任务分析

大部分参与调查的新手们都把很大一部分的时间花在了沟通上。沟通包括会议（组织上的和自发的，由不同数量同事开的会议）、认识和了解团队成员（以及他们的代码和工作内容）、寻求帮助、接受帮助、帮助他人、说服他人、和他人进行协调工作、寻求反馈（如对于代码的反馈）和找人。

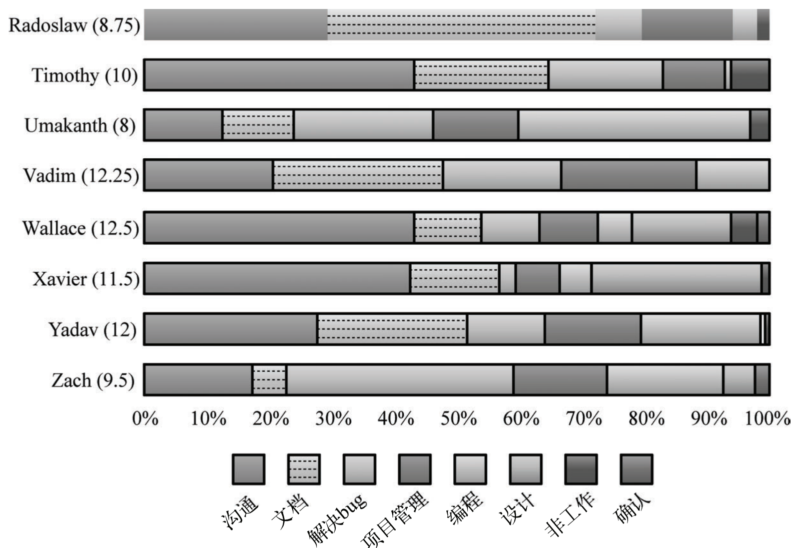


图26-1　研究对象们花在各个事务上的时间，按照总时间+重叠时间来规范化，在每个被观察者名字后面的括号里的是总观察时间（以小时为单位）

1. 沟通

Wallace、Xavier和Timothy花费了绝大多数的时间在沟通上。他们都参与了多个会议并在解决bug的过程中寻求了他人的帮助。Wallace的沟通时间非常多是因为他所属团队的沟通非常容易——他在一个“牛棚”（bull pen，大型公共办公室）式的团队内工作，和其他4个开发人员一起。大家不但在平常时候常常直接以“说”的方式寻求帮助解决问题，而且还在出现紧急情况的时候也这样做。

Xavier以及Yadav两人与其他人相比花费了更多的时间在沟通上，不过除此以外他们还分别在另外的任务上花了很多时间。Xavier花费了大量的时间来和他的团队中负责功能的领导以及另外一个产品的团队开会讨论设计问题。他还经常和团队中的其他新人一起做结对编程和结对调试。Yadav也花费了一些时间在bug优先级排列会议上，但更多的时间是用来是用来写代码，这个时间通常还包括阅读文档来学习写代码的方法。此外，Yadav常常使用即时消息工具作为和团队中其他有经验的成员沟通编程方面问题的工具。

对Radoslaw来说大部分的沟通时间是用来做协调。Radoslaw的任务要求他和多个队友一起分析和协调进行基准测试，并将收集到的结果做成报告。测试用机器的数量有限，需要他做很多协调工作。同样需要协调的还有共享测试结果报告的开发。

总的来说，沟通对于新手是既重要而又常规的一件事。他们用沟通来寻求帮助，并用沟通来学习和理解他们的团队的工作方式。另外，我们所列出的新手所做的任务列表可能还不足以强调沟通的重要性。如果新手花更多时间在和同事的沟通上，他们可能就能更有效地获取信息了。

2. 文档

对于Radoslaw、Timothy、Vadim和Yadav来说，第二常见的任务是和文档相关的事务。在编程和调试中，文档相关的任务包括搜寻、阅读和理解文档（通常是网页形式的API文档），为的是领会旧代码并学写新代码。新手还需要写bug报告和规格计划等文档。最后，新手常常需要自己保存一份文档来记录项目的管理和工具的情况，或者用来记录他们需要在会议中展示的东西。多个研究对象向研究人员提出，保持和整理这些个人记录是很费劲的事情。他们缺乏管理这些记录的规范，并希望在管理这些记录的过程中得到一些帮助。

Radoslaw 和 Vadim比其他人在文档上花费了更多时间。两人的时间都主要花在了读文档上，包括来自文件、MSDN、网页和电子邮件的文档，以帮助他们了解他们团队的代码。这是其他工作（比如写代码，或者安装/建立开发环境）的连带结果。Vadim认为把时间花在这样高强度的阅读上并不十分划算，但是作为一个新人，他也没有别的途径来学习了。

3. 解决bug

第三常见的事情就是解决bug了。不同开发人员之间调试和除错的频率差别非常大，因为这个由每个开发人员所处的开发阶段而定。我们发现Yadav和Timothy常常参加bug的讨论会议，而Wallace的团队虽然有实体和虚拟的bug研讨会，但是并没看到他参加。我们发现每个参加者都曾经重现过bug（有时仅靠他们自己），并了解过这些bug发生的原因。Zach非常专注于调试和除错，并认为减少错误就是成功。甚至曾经有一段时间，他依靠模式匹配修改的方法每天都批量解决几百个bug。我们观察到的其他的调试任务包括代码的测试、浏览和搜索。

4. 编程

有三个新手（Umakanth、Yadav和Zach）都在添加新功能上花了时间，其中的任务包括编程、规格文档和调试。写代码和学习代码的开发人员常常也需要参与文档的制作来理解代码和API是如何工作的（有时也靠模仿在线的代码例子来学习）。尤其是为了支持调试的工作而编程的时候，我们发现新手们会在代码中浏览和搜索——一个对他们来说比较难掌握的过程。最后，我们注意到新手们都非常尽责地在写注释。有时候，他们甚至会在读代码解决故障的过程之中给几行无需修改的代码也加上注释。他们在努力地学习和熟悉代码的过程中非常喜欢注释，而且他们也很愿意添加注释。

Umkanth的第一个项目是设计和实现一个程序向导对话框。这个对话框的实现基于他从没用过的API。他大部分的时间都花在了编程上，但主要是因为他选择了一个低效的学习方法，即一切都靠自己尝试。如果他花多一点的时间去寻求帮助而不是自己一个人研究代码的话，可能会更有效率，也不至于那么郁闷。

Xavier的工作基于去年夏天的一个实习生所写的代码，而这个实习生现在已经离职了，无法回答关于代码的问题。Wallace之前在一个网站服务公司工作。在那里的工作需要持续不断地添加各种小功能和修改各种小bug。由于Wallace之前没见过这样的代码库，他经常问同事很多问题。此外，Wallace还和其他4个在类似项目中较有经验的开发人员一起在一间公共办公室里工作。他们每个人都可以回答Wallace遇到的问题（有时也会问Wallace问题），Wallace不用专门去找某一个人问。

5. 项目管理及工具

我们观察到，新手们会做很多项目管理及工具相关的事情，例如使用版本控制系统，编译他们的程序，安装开发环境，运行辅助工具和创建一些用于支持项目管理和流程的小工具。项目管理和工具相关的事务似乎占用了新手们过多的时间。更重要的是，这些事务往往会打断他们的工作。一个极端的案例是，我们资历最浅的研究对象（Vadim）因为项目相关杂务的牵绊而完全没办法开始配置开发环境。还有Zach，他所在的团队当时正处在一个修改bug的阶段，而他分到的任务则是解决1300个新的编译器警告。他把每一批解决的编译器警告分别存为不同的检查表，而每个检查表都需要和同事一起协商进行代码审核再提交到版本控制系统。

6. 设计规格文档和测试

在设计上花费的时间主要取决于新手开发的项目所处的阶段。Xavier和Wallace都花费了大量的时间来理解和编写设计文档。Zach所在的项目中测试花费的时间尤其多。其他的几个新人也在测试上花了一些时间。

26.3　新手开发人员的优点和缺点

我们的观察结果揭示了新手开发人员的各种优点和缺点。他们的强项包括：

- 编程；
- 阅读和编写规格文档；

- 调试（执着并且能尝试寻找原因）。

缺点包括：

- 沟通；
- 意识；
- 适应性（适应大的代码库以及和现有团队的合作）。

26.3.1 优点分析

我们通过观察发现，这几位软件开发新手都在大学里面掌握了很好的编程、设计和调试技能。他们会用集成开发环境（IDE）来写代码，用命令行的工具来分析文件，并利用在线文档来学习不熟悉的API。他们不只是纯粹地完成任务，还想弄明白为什么代码应该这样修改。他们中的一些人使用了“复制粘贴大法”，从各类文档和同组的其他人写的代码中把需要的部分粘贴过来。在调试的时候，所有的新手们都曾积极地试着自己寻找出错的原因（参见26.1.3节），并坚持尝试所有的可能解决方法，如果不行才放弃。他们中大部分人在解决问题的过程中，都耐心地经历了一次又一次地失败。虽然他们也自言自语地说一些类似于“不熟悉工具啦”或者“没有使用说明啦”一类的抱怨，但是却很少把这个想法告诉给其他人。

有一个新手的工作是要检查软件的两个功能的设计文档。这个任务带来的影响就是他需要频繁地和组长进行讨论，进而需要进而澄清设计的细节以及阐述具体的使用案例。这个经历让他在第二个月中过得比较舒服，在此期间他独立写了冗长的结构化开发规格文档。这里我们不得不提到这个组长出色的指导能力。他非常开放地回答各种问题，指导组员的工作，传授以往学到的关于代码和设计工作的一些背景知识，而且对于目前这个阶段的设计文档需要包括的内容他也提出了自己的建议。

26.3.2 缺点分析

我们发现新手们在受观察期间在沟通、合作、意识和适应性上遇到了许多问题。

新手们的沟通问题中最突出的是不知道何时及怎样来问问题。一般来说，新手问问题并不及时，而且也不知道应该问到什么程度。有时候他们在设计会议中太过于注重细节。而在问问题的时候，他们又问得太笼统，而且在别人回答自己问题的时候也没有完全弄明白，往往最后导致误解。

我们留意到的社交方面的问题主要集中在在大型团队中工作、与多个团队进行合作以及适应大型的现有代码库上。很多时候，新手们被明确告知：书面文档几乎没有，而且原来负责这个功能的开发人员也已经离职了。这通常意味着类似于“（要想要文档）只能靠自己”和“以后的日子会很痛苦，因为可以问的人也没有”。

团队和学校在规范上的区别让很多新手感到困惑。Timothy吃尽苦头后才明白了他所在团队的开发流程。他曾经修复了一个bug，并提交等待检查。然而，他的这个修复被驳回，最后没有被包括到发布的程序中。而且被驳回的原因并不是他所预想的代码质量问题，而是经理们（在另

一个组）没有时间来批准这个修复。虽然Timothy认为他可以为客户解决一个bug，但这次为了“不添乱”，他仍然乖乖地接受了经理们的决定。他认识到开发人员的工作不只是编程，在某些情况下，他们还必须在工作中推销自己的代码和思想。一个月后，Timothy再次面临了这个问题，但这次他的准备就充分多了。在给bug的修复排优先级的会议上，他井井有条地阐述了他发现的bug的情况，并详细说明了他解决bug的过程，以促使经理们批准。

新手们费劲地收集、整理了他们需要学习的文档，并做好记录。有一个新手回忆到自己做笔记的情形，说：“笔记总是很乱，我自己常常都不明白我写的是什么。真希望我能有一个更好的方式来做笔记。”即兴的小课堂（比如在新手的电脑前指导一下如何使用版本控制或者bug数据库）通常都没有什么条理，在词语的选择上一般也没有考虑到新手的背景和文化情况。在这类情况下，“老师”们通常像连珠炮一般迅速地就讲完了，使得新手常常怕浪费老师的时间而不敢打断老师来问问题。而结果就是新手们所学到的东西常常是毫无系统性，而且零零碎碎的。

新手们难以适应项目团队、代码库和各种资源造成的信息不足的环境。不过，有时候这和组织混乱以及管理不善的文档有很大的关系。对于新手来说，这样的文档难以被高效地浏览和使用。而新手还在另一个问题上挣扎，那就是“什么时候我该不知道”。因为要学的东西太多了，他们慢慢习惯了对于工具或者代码一知半解的状态，并尝试自己独自学习。虽然现实的确如此，但是这样的习惯使得很多新手在确实应该寻求别人帮助的时候却认识不到。在我们的研究对象中，资历最浅的新人常常出现这种情况。即使在问了问题，然后得到了一个含糊不清的答案的时候，他也仍然接受并据此钻研下去。

有些新手悲哀地感觉自己和团队有隔阂，有几位甚至连自己团队的成员都认不全，也不知道某一类问题出现的时候应该找谁（或者知道找谁但是不知道他的办公室）。这严重地影响了他们的效率，并让他们感觉到巨大的挫败感。在与团队的隔阂度这一点上，各个新手之间的差别很大。有一个新手选择了从另外的渠道了解信息，然后花了大量时间来阅读各类高级文档，希望能了解他所在团队的工作情况（最后基本徒劳无功）。另一个新手发现最容易了解信息的方法还是和人交流。他没有去搜索各种文档，而是到处去找同事问问题。如果要找的人不在或者不知道答案，他就再问下一个。只有当知道答案的人不在办公室的时候，他才会退而求其次地选择效率低一些的学习方法，比如看代码或者做一些测试。

26.4　回顾

虽然我们的观察记录客观地总结了新手们做的事情，但它并不能囊括新手们感受的所有层面。所以，我们需要知道新手对于他们工作的感悟。在这一节中，我们分类整理了一些新手做视频日记时的一些回顾。这些回顾将帮助我们更深入地了解社交和层级因素对于新手感受的影响。

实践证明，预先在日记中提供的问题可以引导新人们反思他们在大学和在行业中学习的感受。我们在这里列出几个回答最多的问题。

- 你的大学经历对你在微软的工作提供了多少准备？
- 高校毕业生们如果想过好在微软的头一年，最应该做好哪些准备？

- ❑ 如果“未来的你”回到现在，你觉得他会教给你什么？
- ❑ 如果你能穿越时光见到刚刚度过在微软第三个星期的你，你会给他什么建议或者忠告？
- ❑ 你怎么知道你真的陷入困境了？这种情况下你又怎么办呢？

在接下来的几节中，我们将分析和总结新手们对这些问题的回答。

26.4.1 管理层的介入

新手们在刚刚入职的时候立刻就会面临一个窘境。他们觉得必须要向他们的经理证明他们多么聪明、多么高效、多么可靠——即便是在他们还没完全搞明白他们所在职位具体工作的情况下。然后，由于他们想要在短期内一步登天地学习很多东西，并都想选择那些能出位的任务，导致压力和焦虑水平不断上升。我们的新手们在回顾中都提到了这一点，而且大都会后悔当初想一口吃成个胖子的学习态度。Timothy说：“你不需要去深入了解某一个部分，但是你需要对每个东西都有一定的了解——太专了不行。钻牛角尖会让你花很多时间，还会延误任务的进展。”Vadim认为他应该花更多时间来学习：“在刚刚进入公司的第一年，在不工作的时候应该多用一些时间和精力在学习新技术和深入了解产品上。”Wallace回忆了他在精神上的高压：“我可能会告诉当初的自己不要有太大压力，要冷静地对待各种事情，因为压力对工作没任何帮助。反正你在学习，我也在学习，大家都在学，我不想在这种细节上太纠结了……”Xavier提到了一个大家在初入职场的时候都不愿意采用的方法：“多向他人问问题。”“你自己需要花一整天来看代码和文档来解决的问题，问别人的话可能5分钟就解决了。”Vadim这样说道。经常问问题的话，同事和领导们就会知道你知识上有欠缺，这一点对于很多新人来说很难接受，因为他们认为这样会让领导觉得招错人了。

26.4.2 毅力、疑惑和新人特质

Perkins等人将新手程序员分为“浅尝辄止型（stopper）”和“锲而不舍型（mover）”两种[9]。浅尝辄止型新人很容易就会陷入困境然后就放弃了。锲而不舍型的新人会尝试、修正、尝试、修正，然后一直坚持到问题解决为止。我们所有的研究对象们都注意到了毅力的重要性，有毅力就容易成为锲而不舍型的新人。Wallace特别提到一点，那就是“在微软有一种不言放弃的态度……如果是我提出的问题，那么就该我来解决。去上司那儿说‘这个我搞不定’是行不通的……说到底，这还是我的责任。”

Perkins提出，对于任务不确定和对自己能力不自信会使学生们变成“浅尝辄止型”。不过，我们的观察记录显示，即便是非常有毅力的学生也还是显示出了对任务的疑惑和对自己能力的怀疑。当问到关于这类问题时，我们的新手们都没有承认他们有浅尝辄止型的特点，不过Wallace说道：”我经常不知道我其实已经钻入牛角尖了，该寻求帮助了。但我还是闷着头不断尝试直到找到解决方案为止。”所以，锲而不舍并不意味着就会成功，有时候甚至正好相反。正如Xavier说他被困难卡住时候的情况：“当我已经尝试过所有已知的方法但是还不能解决问题的时候，我才知道我被卡住了。”在被问题困住而无法取得进展的时候，更好的解决方法就是向别人寻求帮

助。但是由于权力上的不对等以及社交上的不安，使他们不愿意采取这个方法。其实，我们观察的新人中的有一部分确实寻求了帮助。Timothy说："当我被卡住的时候，我就去问一个资历老一些的队友，看看他们以前有没有遇到过类似的问题。"但是，多数情况下新手只会在已经手忙脚乱地弄了很长时间，甚至花了好几个小时都还解决不了问题的时候，才去问别人。

由此可能会得出这样的结论，对工作的疑惑和对自己能力缺乏信心等都属于这些新进开发人员的"新人特质"，而不是由学习编程引起的。而事实上，也确实有组织机构管理方面的研究证明了这一点 [1]。

26.4.3　大型的软件团队环境

新人们的回顾表明，他们在软件开发的技术/职能上的准备得很充分。Zach说："我觉得我不需要学更多的技术知识了。"但是，他们的回顾也同时表明，他们在社交方面的准备十分欠缺。Vadim提到："在大学里面我们的团队也就2~3个人，而且项目之间无需合作。不止是我，随便谁也不可能在一开始就准备好和75个人一起合作，其中包括35个开发人员和差不多数量的测试人员，（我还必须）跟他们一一合作。"沟通不畅的后果是可怕的（接Vadim前面说的）："做任何事都得小心翼翼地不去破坏任何规矩或者打扰别人。"Xavier说道："即使一件事看上去很容易，但有时做起来也很难。一个看上去很小的修改，结果可能需要大动干戈。这些有趣的连锁反应和副作用，都是你在专注于某件事情时所想不到的。"与队友紧密地合作是取得成绩的有效途径之一。Wallace说道："我认为我需要的是团队精神方面的提高……当我的队友们取得成就的时候，或者当他们需要帮忙的时候，我希望能够把他们的目标也作为我的目标。因为他们的成功就是团队的成功，而我想让我们的团队更成功。"最后，虽然我们并不知道Xavier在大学的时候有没有机会和更多的人一起参加较大项目的开发，但他表示他还是希望能有这些方面的经验："（我早该）和其他人一起做项目然后学到更多团队合作的经验……而不是自己傻傻地去做那些只有一个星期的家庭作业。"

26.5　妨碍学习的误解

在观察中我们发现，很多时候新手们的一些误解会妨碍他们和同事的交往。

- 我只要独立地完成所有事情，我的领导就会觉得我很牛了。

　　这种错觉是非常危险的，特别是在大型和复杂的开发环境中。我们主要在非美国本土的新人身上看到这一点。他们认为需要"表现"而不要被"揭短"，并把很多时间白白浪费在挣表现上了。这种挣表现的行为似乎也让他们更少地进行沟通，并需要更长的时间才能适应新环境。有时新手会很钻牛角尖地独自解决问题，有时还试图掩盖他认为他"应该"知道但却不知道的问题。在这种情况下，沟通就更困难了。此外，即使队友已经明确告知某个问题应该由别人解决或者根本没必要解决时，有的时候新手们还是会继续在上面浪费时间。虽然我们调查规模非常小，但是有一点仍值得注意，那就是在来自美国本土的新员工身上没有发现这样的问题。

在两个月的观察期中，我们的新手们变得更自信了，更能面对压力了，也收获了自尊。在最后研究结束之前的采访中，参与研究的新手们很多都表示，他们当初的那些担心和设想跟现实完全不同。

- 我发现的bug必须我来解决，而且我应该知道怎么来“正确”地解决它，即使我根本没有时间。

每个新手几乎都有这个错觉。毕业生们在大学里面缺乏以团队为基础的开发经验，又必须适应严格交付期限的打分机制，这些也许是造成这个错觉的原因。新手们有这样一种看法，那就是任何东西只要“用不了”，就必须立刻修复。虽然他们很明白有一套成文的规矩来报告、分级和解决bug，但是他们常常还是图方便，绕开这些步骤来解决问题。有的新手被“抓现行”的时候，受到了批评，但是这个信念似乎已经非常根深蒂固了，可能需要更多的时间才能慢慢改变。

- 如果文档更全的话……

比起误解来说，对于完备的文档的渴望更像是一个很正常的需求。有些稍微资深点的新手同时还认为，就算文档很容易就过期了，他们还是希望越多越好。此外，稍微资深点的新手还希望能得到更多工作环境中其他人的信息，即具体代码和具体问题的负责人是谁。这是因为他们已经意识到软件开发的复杂性和紧凑的时间安排使得文档非常有限，而在这种情况下，人就成了最好的活文档资源。

- 我知道我什么时候开始钻牛角尖并且该找人帮忙。

上面这句话有四个新手（Umakanth，Timothy，Vadim，Zach）说过，但是实际观察到的情况正好相反。很明显，新手们几乎总是在浪费时间、精力和钱在手忙脚乱地解决问题上，但是总是不能意识到他们应该寻求帮助了。这个结果也许不令人意外，因为在计算机课程中并不会教授元认知[①]的拔改。虽然新手们常常被一个问题搞得精疲力尽，但是这似乎也和没有人告诉他们怎么来认识自己的状况，以及遇到卡壳的情况下该怎么办等有关系。值得注意的是，虽然博士生们更倾向于自我审视和反省，但是他们在钻牛角尖方面和本科生没什么区别。

26.6 教育方法的反思

在这一节中，我们将从新人社会化的角度来点评一下各种计算机科学的大学课程设置和教育方法。我们之前提到的软件开发技能，如编程、调试、测试、项目管理、编写各种文档、工具和沟通，等等，在大学里面都有不同程度上与之对应的课程。对于每个技能，我们都能找到Schein所说的与组织相连的三个层面：职务、层级和社会网络。三个相对比较新的不太常见的教学方法，即结对编程、合理的边际参与以及导师制，正好可以在大学的环境下解决新人在结构和社会层面的问题。我们强烈建议学校和公司更广泛地使用这些方法。

---

① 元认知是对认知活动的自我意识和自我调节。——编者注

26.6.1　结对编程

参与研究的新手们都觉得惊讶：软件开发流程中竟然需要如此多的沟通。鼓励沟通的一种方法是结对编程练习。两个学生组成一个二人小组进行开发，相互配合着编辑代码并编写文档。Laurie Williams在本书的第17章中对于结对编程法在实际工作中的效果进行了探索，而结果显示，结对编程法增加了在校生们的效率和对自己能力的信心[6]。由于可以集思广益和交换意见，还可以在实在疲惫的时候将问题交给对方来解决，所以结对编程的参与者们通常都不会感到那么大的压力，也不那么容易烦躁。结对编程还可以让学生们相互熟悉，相互间还可以自由地提问题。

不过，结对编程并没有解决组织结构上的问题。结对编程中的两个人通常经验和背景相同，而且又都是为了得到好的成绩而奋斗。与软件行新人的情况对比一下，就会发现。和学校的情况相似的是，在公司中会有领导对于新人的表现进行评估，而且和大学教授一样，领导也会先给新人足够的时间来学习知识技能，然后再严格打分。但是和大学环境不同的是，应届毕业生们在初入职场的时候并不真正清楚他所在的团队有哪些人，也不知道团队中谁最擅长什么，而且自己还是整个团队资历最浅的人。

26.6.2　合理的边际参与

大学教育普遍有一个问题，那就是和现实生活的脱节。由于行业的发展比起教育要快很多，所以这种脱节性也就一直存在着。Mark Guzadial提出的多媒体编程教学法（Media Computation）正是为了改善这个问题 [3]，让学生们知道他们应该在实践中学习。

此外，Lave和Wenger在他们的书中也有过相关讨论，并提出用拜师的方法来让他们参与到实践团体①中来进行实践和学习。合理边际参与法的知识传播过程是这样的：新手们观察有经验的人进行某个步骤，并随后在他们的指导下尝试进行相同的步骤，熟悉之后再去指导其他人。Guzdial和Tew在他们的多媒体编程教学法的课程中发现，学生们的退学率降低了。即便是在计算机科学课程中更容易打退堂鼓的女性学生的退学比率也降低了。这个课程还改变了对于真实工作的一些错误猜测，让他们结合行业中的实践进行学习，使他们能更好地适应将来的工作。这样的改变大大地增强了新手们对于自己能力的信心，也让他们更愿意留在实践团体再帮助其他人。

Hazzan和Tomayko在他们的“软件工程中人的因素”课程中，详细地模拟了在软件开发团队中的社会动态。他们的软件工程课程教授的内容包括：如何处理在团队中会遇到的各种难办的社交问题、职业道德、在工作中学习的方法和技巧、怎么来理解员工奖励制度、以及影响团队表现和凝聚力的价值观等。

26.6.3　导师制

关于职场新人社会化的研究显示，新人要成功的话，由有经验的导师来进行指导非常重要 [7]。导师可以向自己周围的人介绍新人并帮新人找到可以回答问题的人，还可以在需要时提供帮助。

---

① community of practice，即由一群人通过共同的兴趣、爱好或者技术组成的团体。——译者注

此外，导师还负责向新人介绍设计的基本原理，帮助他们寻找需要的信息，并在流程和方法上给予新人以指导。也就是说，导师将教新人们怎样从社会和技术两个层面融入到团队中来。

对于我们的研究对象中的两位来说，相互指导是一个重要的学习途径。他们是在同一时间加入的微软的朋友。每当他们学到新的东西的时候，就相互分享。这样的学习方法比从导师或者领导传授更加有效。由于相互指导的两个人通常都拥有同样的经验水平，使得他们相互之间更愿意交换信息和知识，而不必担心问太基础的问题会让人觉得自己能力不足。很多大学用导师制来留住为数不多的女性学生和比例更低的少数族裔的学生[5][8]。导师在进行指导的时候会和学徒产生文化和价值观上的联系，这些在日常工作中可能不会出现的联系可以使学徒对于团队更有归属感。这也会帮助新人们从社会网络的边缘朝中心移动。

26.7 改变的意义

我们发现新手们的很多问题都有着相同的根源，那就是沟通的不顺畅以及社交上的稚嫩。根据我们对职场新人社会化的观察，我们提议对企业的“上船”（微软将新人培训称作“上船”培训）和大学计算机科学的课程做一些改变。我们相信这些改变可以让应届毕业生们更好地准备将来的工作，并希望能够帮助他们更快地成长为专业人士。

26.7.1 新人培训

很多新人在加入微软几个月之后就有了导师，而我们也相信在新人进公司的头一个月进行集中指导可以达到最好效果。一个好的导师并不只是简单地跟新人介绍文档、工具、流程、队员等，而是以身作则，亲身为新人展示正确的行为和方法。例如，Vadim有一天找到了他的一个队友（顺便一提，这个并不是他的正式导师），问一个关于重现bug的问题。这个队友是这样做导师的：他和Vadim一起看了bug报告，并发现这个报告写得含糊不清，造成了Vadim的问题。然后他们一起从报告中找到了写报告的人，然后又一起写了一封Email来询问这个报告的详细情况。后来他从公司的通讯录中发现这个人正好在这栋楼，便问Vadim是否想直接去找这个人。这种以身作则的做法可以让新人们感同身受地明白微软的工作规范，如果仅仅只是告诉他们怎么做的话则不可能达到同样的效果。与此相反的是，Vadim的正式导师却很忙，没能帮上什么忙。他经常只是告诉Vadim某某东西或者某某文档在什么地方——而且有时候还会弄错。

我们既然培训管理人员和导师，并给他们时间提升效率和积极性，那也应该给新的开发人员同样的机会。帮助新人们找到和他们资历差不多的人并让他们可以相互问问题（例如每个团队一个新人邮件列表或者新人聊天室），可以使新人的焦虑水平降低，并可以在他们中间形成一个新人团体，使他们可以相互询问和回答一些一般人看来太“愚蠢”的问题。一个类似的做法是用有一点经验的新人来指导完全没经验的新人。他们都是新人，所以对于工作前景的认识都差不多，而且刚刚经历过适应阶段的新人导师们更能掌握如何来指导新人，这些是那些有经验的老员工们做不到的。

认识团队成员以及了解他们在项目中的位置这样一件重要的事情却常常是由新人们自己默

默完成的。我们提倡使用“功能面谈会”的形式把这个流程摆到台面上来。新人每周预约一个不同的开发人员进行一次面谈，从而得知这个开发人员负责的功能及其整体架构、在整个系统之间所处的位置、设计上遇到的困难，还有这个开发人员的工作理念以及他认为他工作中有意义的东西，等等。这样的面谈会可以让新人们慢慢地从软件开发人员的角度（而不是从死板的文档的角度）来解读这个软件系统，也让他们有时间来消化所了解到的信息。同时，面谈会还可以使得办公场所的价值观和文化得到更好地推广，让新人们知道团队推崇什么价值观，排斥什么价值观。

那么，领导们如何来衡量新人的表现呢？他们不会像有经验的开发者那样高效，而且学习也是他们发展的重要组成部分。我们认为，要衡量新人的表现必须要看学习的时间、是否敢于尝试以及和他人合作的情况。此外，我们也应该衡量新人导师的表现（例如可以看新人是否能迅速跟上团队步伐），而不是把对新人的培训当作是一个无关紧要也不会被打分的事情。我们采访到的许多新人提到，他们应该在进入公司最初的几个月只对系统进行多方面有深度的学习，而不是一开始就马上像一个老员工一样进项目工作。他们说，一开始领导并没有要求他们要有多高效，但是慢慢地他们就必须使用很多时间在“实事”上面而没办法花时间进行学习了。我们正在设计一套给团队和导师们使用的评测准则。准则将列出个人发展的一些成就，如实现第一个功能、审核别人的代码、被指派修复一个bug、撰写第一个bug报告等。每个步骤都可以写下来，最后组成一个“课程表”，其中的每一个步骤我们都可以进行监督。这些监督数据可以让公司用来评测对入职培训的改良，看看改良能否有效改善现状。

个人软件流程（Personal Software Process）可以用来帮助新手认清他们的任务完成情况，并生成可以和导师、领导、同事分享的数据，以便及早发现新手在流程上的错误和说明可能的改进。公司应该鼓励新人多思考下一步该做什么，什么时候应该找人帮忙，是不是在困住了，等等。可以使用智能教学系统中监察成绩的方法来帮助我们更直观地认识开发人员的行为，并在新手们应该找人帮忙的时候告诉他们。

26.7.2 学校教育

在很多大学中，新型的软件毕业设计课程（capstone courses）让学生可以接触到完整的设计、实现和测试流程，并让学生在大学环境中学习如何与多人团队一起开发较大型的软件。学生们会担当各种角色，比如需求工程师、开发人员、测试人员、文档管理等，并协同合作，最后向客户交付软件。通常学生们是不分等级的，团队的实际领导者是教学助理或者课程讲师。但是我们的研究显示，现实的工作环境和新型课程所提供的人为的环境有相当大的不同。我们发现的很多社交和沟通方面的问题（尤其是在Xavier和Wallace两人身上）的根源在于工作在基于老代码的大型团队中开发大型软件时的焦虑。还有作为资历最浅的团队成员所造成的焦虑，可能会让人产生“必须要让别人知道我很行”的想法。以及对代码完全不了解所造成的忧虑让人觉得需要“浪费时间”来学习代码。当花太多时间来解决问题的时候也会造成新手们的焦虑。虽然也有教授团队合作的课程，但这些课程通常不会解决这些焦虑问题。

比新型的课程更有建设性的做法，是提供给学生们一个现有的代码库，并要求他们修改（真实或者故意造成的）bug和编写新功能。如果再加上管理的因素那就更好了，比如学生们必须和

更有经验的“同事”（即之前参加过这个课程的学生，还可以作为导师）和“项目经理”（教学助理）来互动。“同事”和“项目经理”可以帮他们了解学习代码库，还可以要求他们反复解决bug直到找到“正确”的解决方案。在开发过程中，学生可能被要求在bug数据库中记录所找到的bug、摸索出bug重现方法，还可以根据某个版本发布计划来确定各个bug的修复优先级。

在大型软件项目中，修复bug不只是修改代码。对于一个bug来说，修复的方法很多，每个方法都必须先通过一次人的检验，再通过一次技术检验。可以试想：这个修复的规模有多大？牵涉到多少行代码？牵涉到的代码越多，产生新bug的几率就越大。这个修复需不需要改动在这个阶段已经封存的代码？如果是这样，那就另找不用动用已封存代码的解决方案。与其根据学生修复了多少个bug来打分，不如让他们只修复和报告他们认为最重要的那25%的bug，并让他们说明这样做的原因。

老师们应该在课程中花时间让学生学会元认知技能。你怎么知道你没走错路？当你问同事问题或者需要告诉他们发生了什么的时候，应该如何组织你的语言？你如何与老师（或者导师）更加有效地互动？当教你的人（有可能是你的领导）完全没有当老师的潜质的时候，你如何学习？如果掌握这些技巧，学生将可以成为更好的同伴导师，帮助周围的软件工程师们更好地适应未来的职位。

26.8 参考文献

[1] [Bauer et al. 2007] Bauer, T., T. Bodner, B. Erdogan, D. Truxillo, and J. Tucker. 2007. Newcomer adjustment during organizational socialization: A meta-analytic review of antecedents, outcomes, and methods. *Journal of Applied Psychology* 92: 707-721.

[2] [Curtis et al. 1988] Curtis, B., H. Krasner, and N. Iscoe. 1988. A field study of the software design process for large systems. *Communications of the ACM* 31(11): 1268-1287.

[3] [Guzdial and Tew 2006] Guzdial, M., and A.E. Tew. 2006. Imagineering inauthentic legitimate peripheral participation: An instructional design approach for motivating computing education. *Proceedings of the second international workshop on computing education research*: 51-58.

[4] [Lave and Wenger 1991] Lave, J., and E. Wenger. 1991. *Situated Learning: Legitimate Peripheral Participation*. Cambridge, UK: Cambridge University Press.

[5] [Margolis and Fisher 2003] Margolis, J., and A. Fisher. 2003. *Unlocking the Clubhouse: Women in Computing*. Cambridge, MA: MIT Press.

[6] [McDowell et al. 2006] McDowell, C., L. Werner, H.E. Bullock, and J. Fernald. 2006. Pair programming improves student retention, confidence, and program quality. *Communications of the ACM* 49(8): 90-95.

[7] [Ostroff and Kozlowski 1993] Ostroff, C., and S. Kozlowski. 1993. The role of mentoring in the information gathering processes of newcomers during early organizational socialization. *Journal of Vocational Behavior* 42: 170-183.

[8] [Payton and White 2003] Payton, F.C., and S.D. White. 2003. Views from the field on mentoring and roles of effective networks for minority IT doctoral students. *Proceedings of the 2003 SIGMIS conference on computer personnel research: Freedom in Philadelphia—leveraging differences and diversity in the IT workforce*: 123-129.

[9] [Perkins et al. 1989] Perkins D.N., C. Hancock, R. Hobbs, F. Martin, and R. Simmons. 1989. Conditions of learning in novice programmers. In *Studying the Novice Programmer*, ed. E. Soloway and J. C. Spohrer, 261-280. Hillsdale, NJ: Lawrence Erlbaum Associates.

[10] [Schein 1971] Schein, E.H. 1971. The individual, the organization and the career. *Journal of Applied Behavior Science* 7: 401-426.

[11] [Strauss 1987] Strauss, A.L. 1987. *Qualitative Analysis for Social Scientists*. Cambridge, UK: Cambridge University Press.

[12] [Taft 2007] Taft, D.K. 2007. Programming Grads Meet a Skills Gap in the Real World. *eWeek.com*, Sept 3.

[13] [Tomayko and Hazzan 2004] Tomayko, J.E., and O. Hazzan. 2004. *Human Aspects of Software Engineering*. Hingham, MA: Charles River Media.

第27章

挖掘你自己的证据

Kim Sebastian Herzig
Andreas Zeller

在这本书中，有很多收集证据的例子，如证明测试效率的证据，说明bug报告质量的证据以及复杂度度量的作用的证据，等等。但是如何将这些证据和结论应用到你的项目呢？要知道这个，最直接的办法就是在你的环境中用你的数据重复适合你情况的研究。这样的话，你不但可以加深对你自己项目的认识，还可以享受到做实验的乐趣。不过，你也可能遇到不好的一面，那就是实证研究的成本可能会很高，尤其是当需要对开发人员进行实验的时候。

好在我们还有一个相对来说花费少一些的方法来批量收集证据。软件的档案（如不同的版本或者bug数据库）记录了很多你的产品的相关事务，例如产生的问题、做了哪些修改以及解决了的问题。只要对这些档案进行自动数据挖掘，你就可以获取关于你产品很多的第一手证据。这些数据不但本身很有价值，而且还可以作为基础，让我们可以进行更深入地试验，进而得到更深入的认识。在这一章中，我们将手把手地交给大家如何对软件档案进行数据挖掘，包括最基本的技术方法以及在数据挖掘过程中可能会遇到的陷阱。

27.1 对什么进行数据挖掘

在软件开发中，程序员们定期性地产生和收集各种数据，可以用来进行自动化分析。这些数据包括如下几项。

- 产品的源代码。这是你需要分析的最重要的数据，因为它提供了各种具体位置（文件、单位、类、组件等），可以用来把产品和流程的各种因素联系起来。
- 收集软件运行的相关数据（即运行情况概略），可以让你知道哪些部分是常用的，哪些部分不常用。
- 你的产品可能还包括额外的文档，比如设计文档或者需求文档，而这些文档也许可以解释代码为什么会是现在的样子。
- 我们还可以对程序进行静态分析，以得出复杂度或者依赖关系。
- 版本档案库记录了对产品做过的修改，包括谁、什么时候、改了什么地方以及为什么这样修改。如果档案库中的修改有清楚的分类，而我们又能系统地分析利用这些修改所对

应的修改原因（rationale），那么我们就可以从版本档案库中了解到项目的很多历史情况。

- 要想将问题对应到位置，那么有一个问题数据库就很重要，这个数据库会记录所有发生过的问题，以及这些问题从发现到解决的整个生命周期。
- 最后，你可能有一些社会学上的数据：开发人员的小组或者团队间的划分、开发人员之间的Email或者其他通讯，甚至是开发人员的收入情况或者工作量的相关数据。有了这些数据，你就能知道每个人在系统的各个部分或者任务上耗费了多少精力，或者每个团队做了多少修改，犯了多少错误。不过，在你将这些团队按照错误的多少来排名之前，请记住，任务越难，错误就越多。经常出错有可能只是因为做的是最难的工作——而最难的工作通常是由最有经验和最可靠的程序员来担当的。

从研究者的角度来看，这些数据的优势在于它们不偏不倚——这些数据客观地记录了各种更改、问题以及相关的信息，并真实地反映了开发人员在处理这些问题时所做的各种事情。不过从另一方面来看，这些数据也有可能是杂乱和不完整的，所以我们需要在分析它们之前进行一些特殊步骤。

27.2　设计你的研究

一旦你确定了想要的数据，你就必须选择一种方法（或者制订一个计划）来进行你的研究。本书已经包含了很多成功的研究方法以及如何进行调查的一些技巧。

不过，在这里我们有一个忠告。虽然很多数据挖掘和研究项目的理念都大同小异，但是如果这种“小异”太多的话，也会使你的研究工作走进死胡同。这种“小异”的因素包括项目的性质，以及项目所使用的开发流程。一个很好的例子是开源软件（如Eclipse）和商业软件（如微软或者SAP）的项目之间的区别。各个软件项目在工作环境以及组织环境上的差异，会使得它们的开发流程产生根本性的不同。比如，在开源项目中开发人员常常来自世界各地，而且大多在不同的地点进行开发。在这种情况下，结对编程或者集体审阅代码就变得很困难，甚至不可能了。即使只是想要简单地就一个问题谈一谈，也必须有先进科技的支持，而且还必须考虑时区的因素。

开发环境和流程的不同会给项目的历史造成巨大的影响，所以我们在对项目的历史做数据挖掘的时候，必须将这些因素考虑进来。从最近的一些研究对开源以及商业项目进行的数据挖掘来看，两种项目各有与之对应的方法[2][16]。很多与流程相关的度量法（如社会-技术网络度量法①[2]）以及组织结构类的度量法[11]的结果都在很大程度上取决于实际采用的开发流程。即便是对于同一款开源软件，这些度量法产生的结果都大不相同。

所以我们建议，在你设计你自己的研究之前，你应该试试在你的项目中复制一个已经证明有效的研究。如果你最后的结果和原来的研究一致或者更好，那么就没问题。但是如果你的结果不同，那就需要调查一下是什么造成了这种区别，因为在你设计自己的研究的时候也需要认真考虑这种区别。

---

① Socio-technical network metrics，简单的说就是一种将软件开发中人的因素和软件系统的因素结合起来（形成一个网络），用于预测下一版本中最有可能出错的组件的度量法。——译者注

27.3 数据挖掘入门

现在我们来介绍一下如何得到真实的历史数据。每个软件仓库都是不同的，所以挖掘有关数据的步骤也是不同的。但是大部分数据挖掘工具的使用方法都差不多。我们将在这一节用一个真实的例子，IBM Eclipse项目，来逐步展示如何对一个软件仓库进行数据挖掘。Eclipse是一个开源软件，而且长期以来一直是很多软件工程研究项目的研究对象。

因此，用Eclipse来作为例子非常合适。虽然关于bug及其修复的数据都是公开的，但是却并不是用系统的方法来收集的，这将让我们的研究很有趣，也很具挑战性。

历史数据以多种形式存在于不同的软件项目之中，如版本控制、bug跟踪系统、Email通信等。为了从软件项目的历史中提取数据并从中学习经验教训，我们必须得访问这些资源。对于很多开源软件（如Eclipse）来说，这些资源中的大部分都是向公众开放的，而且可以随意查看。

接下来我们将详细地展示各个步骤，即如何一步一步地提取Eclipse的历史、流程和bug数据，以用于缺陷预测或者流程分析。一般来说，这些数据将被储存在永久性数据储存系统（如关系数据库）中，使我们可以做更深入的分析和检查。最后，我们将能得到一系列的数据，而这些数据将告诉我们代码中的哪些修改带来了新的bug，还可以让我们知道一共修复了多少bug以及是在哪个文件中修改的。

27.3.1 第一步：确定要用哪些数据

Eclipse在其版本控制系统（CVS）和bug跟踪系统（Bugzilla）中储存了很多关于故障及补丁的信息。还有很多信息可能都被淹没在Email存档里了，但是这种信息毫无结构可言，无法用于统计分析。不过我们即将看到，即便只是对版本控制系统和bug跟踪系统的数据进行挖掘，也还是有很多困难。

在我们这个Eclipse的例子中，我们将源代码的变更历史以及bug报告保存在一个MySQL数据库里面。也就是说，我们必须确定从这些资源中找到的信息有哪些是和我们的研究目的相关的，还必须确定如何来将两个系统中的信息联系起来。我们的目标是找出对源代码做出的哪些修改导致了bug，并找出对应的源文件。为此，我们需要所有能够得到的源代码操作记录（尤其是提交信息）以及bug报告，用于帮助我们确定两点：第一，在哪一次操作的时候这个bug被修复了；第二，修改了哪些文件。图27-1展示了一个数据库表的例子，这样的结构允许我们将代码的变更（`rcs_transaction`）、源文件版本（`rcs_revision`）以及bug报告（`bts_report`）联系起来。

要想把bug对应到文件，我们需要有一个把bug报告和源代码操作联系起来的方法。由于在Eclipse的开发中，两个数据并不是联系在一起的（这一点和新一些的编程环境不同，如IBM的Jazz[6]，就把bug报告系统和版本控制系统联系起来了），我们需要参考开发者的数据输入。开发人员可以输入数据的两个位置分别是*提交信息*（把修改提交到版本控制系统时可以附加的信息）以及bug报告的说明/讨论。我们必须从这两个位置中提取数据。

要想把故障对应到文件，我们必须跟踪在各个操作中源文件的变化。有了这个信息，再加上bug到操作记录的对应，我们便能够找到bug对应的文件。

版本数据库中，对于每一批同时提交上来的代码变更，都有一个变更记录（称为操作记录）。这些记录包括修改人、修改时间、修改的文件以及这些文件修改的部分，再加上一个操作记录。所以，分析这些记录将使我们得到一直以来在这个项目中发生的代码变更的所有历史信息。

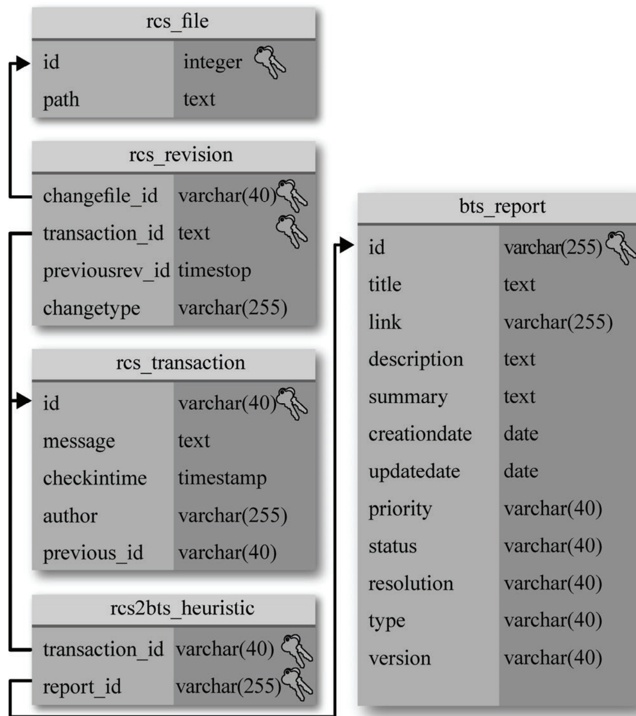


图27-1　bug、操作和文件的数据表结构。箭头表示外键（foreign key）以及连接点

27.3.2　第二步：获取数据

为了保证可以不间断地迅速访问版本控制系统和bug追踪系统，你应该下载一个副本到本地。

Eclipse的版本控制系统有一个已经准备好的转存包（dump）放置在其网站上以供下载[①]。这个转存包可以用来在本地建立一套完整的版本控制系统，包括所有的源代码文件以及所有在Eclipse历史上曾经发生过的源代码修改。

不过要把bug追踪系统也做一个本地副本就要复杂一些了。要分析bug报告，我们必须有一个机器可读而且定义良好的数据格式。Bugzilla符合这个要求。我们可以将单个bug导出为XML文档。要利用这个功能需要用到一个脚本，然后你便可以将所有bug以XML文档的形式下载到本地硬盘，再做其他处理。

①http://archive.eclipse.org/arch/。

下面这个Bash脚本将下载ID为1至300 000的bug报告的XML文档，并保存至一个临时目录：

```
for i in {0..300000}; do curl -o /tmp/eclipse_bug_${i}.xml
"https://bugs.eclipse.or/bugs/show_bug.cgi?ctype=xml&id=${i}"; done
```

有一点需要注意：有一些ID是无效的，因为有的bug报告已经删除，而有的bug报告仅供内部人员阅读。这意味着下载下来的XML文档中有一些会只包括一条错误信息。在分析报告的时候，我们需要记住这一点。

27.3.3　第三步：数据转换（可选）

数据挖掘不止需要你下载和处理数据，还需要你对数据的特征预先有了解。任何时候，当你在对一个储存库进行数据挖掘的时候，都必须牢记数据挖掘的目的：提取项目和流程数据。所以你得到的数据应该符合你所设置的前提条件。

在我们这个案例中，Eclipse的版本系统和bug报告系统中用来作为基础的信息是不同的。bug报告包含的是某个bug修复操作的全部信息。而版本控制系统则不同，它的修改记录所对应的是源文件，而不是操作。所以在Eclipse的这个案例中，将数据资源相互转换格式是很有必要的。为此，你应该将CVS的数据转换为一个以操作为基础，而不是以文件为基础的格式，例如Subversion（SVN）。要做这个转换，我们可以使用cvs2svn这个工具[①]，它可以保证数据不会在转换过程之中丢失（包括GCC、Mozilla、FreeBSD、KDE、GNOME等项目的CVS库都可以用这个工具来转换）。

下面这条命令可以把Eclipse的CVS库转换为SVN库：

```
cvs2svn -svnrepos path_to_new_svn_repo -include-empty-directories -no-prune -no-cross-
branch-commits -retain-conflict-attic-files --verbose
```

注意根据CVS库的大小，转换将需要一些时间。为了确保顺利完成转换过程而不出现错误，我们强烈建议在正式转换之前，先给刚才的命令加上`--dry-run`选项试运行一次。加上这个选项之后，转换器就不会真的转换数据，而只是模拟转换过程，并把实际转换中可能遇到的问题告诉你。

27.3.4　第四步：提取数据

到此，你在本地硬盘上的数据已经准备好了，并且这些数据也符合前提条件，接下来可以开始理数据了。数据的处理包括提取、过滤以及将这些数据以人类可读的格式保存于永久性储存设备。

在这一步我们用版本控制系统和bug数据库中得来的数据生成XML文档。我们前面分析的，Eclipse的版本控制系统中包含的是操作相关的信息。我们可以使用下面这个SVN自带的命令将这些信息提取为XML文档：

```
svn log -xml --verbose
```

① http://cvs2svn.tigris.org。

你可以用你喜欢的XML解析框架来解析这些XML文档以及在第二步中下载下来的bug报告，然后，将解析出来的数据存在数据库内（见图27-1）。

27.3.5　第五步：解析bug报告

下一步是对在第二步中获取的bug报告进行解析。填好`bts_report`表的所有字段是很重要的，特别是`ID`、`title`（标题）、`description`（描述）、`creationDate`（创建日期）、`status`（状态）、`resolution`（解决方案）和`version`（版本）。我们可以用这些字段来把bug报告和版本控制操作记录联系起来。

27.3.6　第六步：关联数据

我们已经从数据资源中提取了所需的信息，现在要做的是把这些数据相互对应起来。到目前为止，所有数据都是独立收集的，没有和其他的数据进行相互关联。

要将bug和操作对应起来，你必须依靠那些直接或间接地包含了交叉参照（cross-reference）信息的用户输入。在我们的Eclipse案例里，这种数据可能从两个渠道得到。

- 当提交代码修改到版本库的时候，开发人员直接指出了所修复bug的编号（如：“修复了bug 88352：操作导致环境菜单泄漏……”）。
- 当关闭一个bug报告的时候，开发人员提到了修复这个bug的操作记录ID（如“修复了Windows版的问题，SVN版本6800。麻烦谁再检查一下……”

你必须在操作提交信息以及bug报告两者中都搜寻可能的相互对应关系。

就算找到很多错误的对应关系也很正常，你需要随时根据当前结果准备调整你的规则表达式。你可以反复地进行搜索，然后亲自检查每一次得到的数据集中的问题，用于改善下一次搜索。

1. 将代码变更和bug报告联系起来

我们发现，最常见的情况是开发人员把修复的bug ID写到提交信息里面去。Fischer 等人[5]以及Cubranic和Murphy [3]是第一批在更改提交信息中搜寻bug数据库的引用信息并用这些信息来找出CVS库和bug数据库之间联系的人。此后，Sliwerski 等人改良了识别解决bug的代码修改的方法，这种修改存在于已成功解决了的bug中[13]。依照他们的方法，我们需要执行下列步骤。

(1) 选择所有包含可能有修复bug意味的关键词（如“修复”、“问题”、“bug”、“解决”等）的提交信息。这些提交信息有可能包含和bug数据库交叉的参照信息。

(2) 在这些提交信息中用以下规则表达式进行搜索①：

- bug[# \t]*[0-9]+
- pr[# \t]*[0-9]+
- show_bug\.cgi\?id=[0-9]+ or \[[0-9]+\]

每个匹配结果都有可能和一个bug相关。不过，这个方法也有可能给你很多误报，如“Updated

①此表仅供参考，并不完善。

copyrights to 2004”（将版权信息更新到2004年）这样非的bug信息也会被找出来。此外，如果想要知道找到的bug报告链接是否有效，那么你还必须检查链接那一头的bug报告信息。由于你需要了解的只是那些真正解决了bug的代码修改记录，你可以只关注链接到下面几种bug报告的记录：

- 状态标记为“已关闭”的；
- 解决情况标记为“已解决”的；
- 解决日期和更改提交日期接近的（如相差在5天以内）。

还有很多其他的条件可以用来判断链接的有效性，但是这些条件通常和软件项目的开发流程有密切联系，必须针对不同的项目进行调整。还有一点需要记住的是一个操作所修改的代码有可能解决多个bug。

2. 将bug报告和代码变更联系起来（可选）

虽然用扫描提交信息来搜寻交叉参照的方法对于很多项目而言已然够用，但还是有一些项目需要用其他的方法。Cubranic和Murphy是第一批为这些链接做反向联系的人（从Bugzilla联系到CVS库），方法是将bug相关的行为和代码的变更联系起来[3]。

如果有需要，我们可以将前一步骤中的流程反转，在标记为“已关闭”和“已解决”的bug报告中搜寻可能的版本ID和关键词，如“revision”（修正版本）、“transaction”（操作）、“patch”（补丁）等。然后，试着将这些ID和操作ID对应起来。和之前相同，我们还需要过滤误报，即删除在bug标记为“已解决”之后才应用的版本ID。同样需要记住的是，一个bug报告也可能联系到多个操作。这些操作中有的可能是早前不完整的修复。而其他的操作可能已经被取消了。所以，添加引用的顺序可能也有一定重要性。

27.3.7 第六步：找出漏掉的关联

在把bug报告和代码变更相互联系的时候，你将会发现有一些bug报告或者代码变更一直没有关联，也即是说，你并没能找出某些bug是由哪次操作修复的，而对于版本控制库中的某些操作记录，你也不知道它们是为了解决什么问题。这种漏掉的关联是数据干扰（noise）的一种。在数据挖掘中是肯定会漏掉一些关联的，但是如果不进行人工筛选分类，这个问题就很难解决。不过，有一点要注意的是，漏掉的关联有时并非随机出现，而是与某些干扰因素有关。例如，有经验的开发人员可能会更加自律地把代码更改和bug报告关联起来，结果就会造成我们观察到的关联也可能只是基于更有经验的开发人员的片面观点。

最近，Bird等人向我们展示了这种片面的数据所带来的严重问题，尤其是在生成bug预测模型以及用bug数据来做假设实验的时候[1]。对数据集进行人工检查不能解决这类问题，但是可以用来检测明显的数据偏差。

27.3.8 第七步：将bug对应到文件

现在你已经将bug对应到修正控制的操作了。接下来，需要把bug对应到源文件。幸好，这一

步很简单。用我们的数据库结构（见图27-1），你可以很容易地找出哪些操作修改了哪些文件。Zimmermann 等人使用了一个类似的方法来计算每个源文件所对应的bug的数量 [5]。

然后我们能得到一组数据，记录了每个源文件中修复的bug数量，这可以作为该文件的质量指标。这个数据可以用来和其他的参数相关联，如文件修改的次数或者文件的代码复杂度等。

例如，在图27-2中展示的，是在Eclipse 3.1中用了超过300行代码的修改来修复的bug的分布情况，这些信息正是我们用本章中描述的步骤从Eclipse的库中获得的①。图中的每个方块都代表Eclipse 3.1版中的一个源文件，来自同一个包的文件放在一起。方块越大，就代表这个版本中修改的行数越多。而方块越白，就代表这个包在Eclipse 3.1版本中需要修改的bug越多。这样的数据带来了很多问题，如：为什么有些类那么容易出错？我们可以试着寻找常见的原因，对将来bug的位置做出预测并检查这些预测，便可以得到一些有趣的见解。当然，在你自己的项目中应用这些技巧无疑是最激动人心的，因为所有做判断所需的数据已经尽在掌握了。

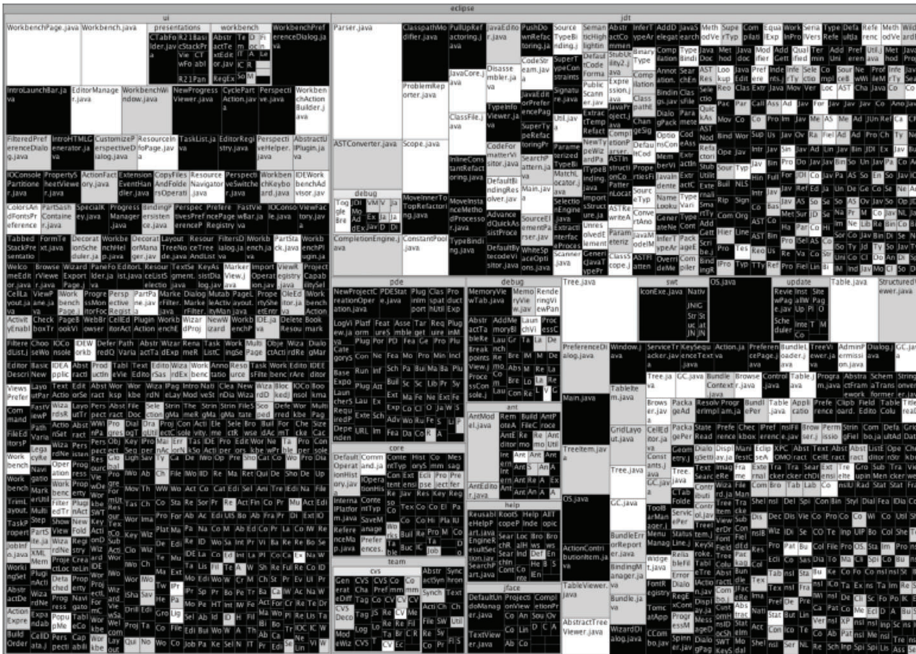


图27-2　Eclipse包中的更改和bug分布。方块（代表包）越大代表修改的行数越多，方块越白代表出现的bug越多

27.4　下面怎么办

图27-2所示的bug和更改分布情况本身就有一定价值：你可以直观地看到被修改最多的包，

① 这个图是用马里兰大学开发的Treemap程序生成的，http://www.cs.umd.edu/hcil/treemap/。

还可以看到问题最多的包。下面要做的，就是找出形成这个分布的原因。作为一名管理者，你应该能解释bug和更改为什么会这样分布，尤其是那些极端情况（如bug最多或者更改最多的模块）发生的原因。不过，你可能还是会对某些不那么极端的问题感到意外：那些遍及整个项目，并且有可能和某些反复出现的原因有关的问题。

到目前为止，根据我们的经验来看，每个项目中的问题根据项目不同而有着不同的与bug/更改相关的成因，这也是我们要对项目的历史进行数据挖掘的原因。我们可以从以下几点开始进行调查。

- 组件的问题范围

  通过分析组件所使用的API，Schroter等人指出，Eclipse内部编译器的代码的出错率是其GUI组件的7倍[12]。这类证据不仅可以用于确定你应用程序的关键区域，还可以在没有历史数据的情况下，观察组件所使用过的API来预测新组件的质量。

- 源代码的复杂度

  越复杂的任务就越容易出错。当源代码变得越来越复杂，要想不留缺陷地进行修改也跟着变得越来越困难。关于如何定义代码的复杂度有很多种方法，而且很多研究显示，代码的复杂度确实可以作为很好的问题预测指标。

- 组件的变更历史

  最近修改（或者有过大量修改）的组件容易产生较多bug。Hassan和Hold介绍了一种故障多发模块缓存算法，可以找出最有可能包含bug的10个子系统[7]。要构建这个前10名的名单，只需使用最频繁更改、最近更改、最频繁修复以及最近修复4种模块的历史信息。不管你是管理者还是开发人员，都可以使用像“前10排名”这样的办法来确定哪些模块最容易出问题，然后重点测试这些模块。

- 组件的测试历史

  组件经历的测试越多，包含bug的可能性就越小。Hutchins等人的研究表明，高覆盖率（超过90%）的测试集比随机选择的测试集更容易检测出故障[8]。后来在Nagappan等人的研究中，也曾成功地用覆盖率数据信息来建立故障预测模型[9]。也就是说，对你项目的测试数据集和系统的健康信息进行数据挖掘，可以帮助你检测代码中可能存在问题，但尚未经过完整测试的部分。

- 在组件的开发中涉及的人

  软件是由人写的，而人是经常出错的。所以，在确定组件的质量时我们也需要考虑源代码的作者。在微软，Nagappan等人做了一个调查，即分布式开发团队所开发的组件是否比本地团队更容易出问题[1]。但令人意外的是，两者并没有多大区别。但如Ekanayake等人的研究显示，编辑同一个文件的人的数量和这些人所修复的缺陷的数量会影响故障预测的品质[4]。

结合这些数据，你会发现一些相关性，但是这些相关性并不意味着因果关系。例如，当我们尝试将Eclipse中的bug密度和每个开发人员联系起来的时候，我们发现首席架构师Erich Gamma居然是bug密度最高的人。而我们问Gamma为什么会这样的时候，他说：“总会有一些问题是你解决不了的，所以你只能交给你的上级，觉得凭他的能力应该可以解决。然后这些问题就这样一步

步传到最上层——先到组长那里，再到部门总管那里，如此这般，而且没人愿意去搭理它们。最后这些问题就都跑到我这儿来了。我没上级了，所以我必须自己去解决这些问题。实际上，我整天做的事情大部分都很高风险。”

记住这一点，你就可以开始挖掘数据中的相关性了，然后再根据你的理论，将这些相关性整理成因果关系。不过要注意的是，用这样的方法得到的证据将会带来更多的问题，也就需要更多其他的证据来解决。一套自动挖掘数据的措施将让你更轻松地获取这些证据，并用这些真实的数据来支持你的判断。

27.5　致谢

本章中描述的多种基本的数据挖掘技巧是和Peter Weigerber以及Thomas Zimmermann[14]共同开创的，谨在此对他们表示永恒的谢意。此外，感谢Yana Mileva、Nadja Altabari和Clemens Hammacher在本章初期修订的时候提供的宝贵意见。

27.6　参考文献

[1] [Bird et al. 2009a] Bird, C., A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. 2009. Fair and Balanced? Bias in Bug-Fix Datasets. *Proceedings of the Seventh Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*: 121-30.

[2] [Bird et al. 2009b] Bird, C., N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. *Proceedings of the 20th International Symposium on Software Reliability Engineering*: 109-119.

[3] [Cubranic and Murphy 2003] Cubranic, D., and G. C. Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. *Proceedings of the 25th International Conference on Software Engineering*: 408-418.

[4] [Ekanayake et al. 2009] Ekanayake, J., J. Tappolet, H. Gall, and A. Bernstein. 2009. Tracking concept drift of software projects using defect prediction quality. *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*: 51-60.

[5] [Fischer et al. 2003] Fischer, M., M. Pinzger, and H. Gall. 2003. Populating a release history database from version control and bug tracking systems. *Proceedings of the International Conference on Software Maintenance*: 23.

[6] [Frost 2007] Frost, R. 2007. Jazz and the Eclipse Way of Collaboration. *IEEE Software* 24(6): 114-117.

[7] [Hassan and Hold 2005] Hassan, A., and R. Hold. 2005. The Top Ten List: Dynamic Fault Prediction. *Proceedings of the 21st IEEE International Conference on Software Maintenance*: 263-272.

[8] [Hutchins et al. 1994] Hutchins, M., H. Forster, T. Goradia, and T. Ostrand. 1994. Experiments of the effectiveness of dataflow- and control flow-based test adequacy criteria. *Proceedings of International Conference on Software Engineering*: 191-200.

[9] [Nagappan et al. 2006a] Nagappan, N., T. Ball, and B. Murphy. 2006. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. *Proceedings of the 17th International Symposium on Software Reliability Engineering*: 62-74.

[10] [Nagappan et al. 2006b] Nagappan, N., T. Ball, and A. Zeller. 2006. Mining metrics to predict component failures. *Proceedings of the 28th International Conference on Software Engineering*: 452-461.

[11] [Nagappan et al. 2008] Nagappan, N., B. Murphy, and V. Basili. 2008. The influence of organizational structure on software quality: An empirical case study. *Proceedings of the 30th International Conference on Software Engineering*: 521-530.

[12] [Schröter et al. 2006] Schröter, A., T. Zimmermann, and Andreas Zeller. 2006. Predicting component failures at design time. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*: 18-27.

[13] [Sliwerski et al. 2005] Sliwerski, J., T. Zimmermann, and A. Zeller. 2005. When do changes induce fixes? *Proceedings of the 2005 International Workshop on Mining Software Repositories*: 1-5.

[14] [Zimmermann et al. 2004] Zimmermann, T., and P. Weiβgerber. 2004. Preprocessing CVS Data for Fine-Grained Analysis. *Proceedings of the First International Workshop on Mining Software Repositories*: 2-6.

[15] [Zimmermann et al. 2007] Zimmermann, T., R. Premraj, and A. Zeller. 2007. Predicting defects for Eclipse. *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*: 9.

[16] [Zimmerman et al. 2009] Zimmermann, T., N. Nagappan, H. Gall, E. Giger, and B. Murphy. 2009. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*: 91-100.

第28章

正当使用“复制–粘贴”大法

Michael Godfrey
Cory Kapser

<Ctrl-C>，<Ctrl-V>，虽然每个软件开发人员都知道“复制–粘贴”大法（又叫代码克隆）是个坏习惯，但每个人又都会这样做。我们有时可能会因为这种偷懒的行为而觉得难受，但是最后却还是对自己说这些都是编程的需要嘛，然后继续复制粘贴。毕竟在大部分情况下，先复制粘贴一段差不多功能的代码，再慢慢修改成需要的样子确实比重头写要快些。

但复制粘贴却还是让我们感觉不自在。我们都知道，想停下来不一定真的就能停下来，就像众所周知的美味薯片，谁都明白只要吃了第一片，要想停下来就很困难了。而且，我们也怀疑这个省事的做法可能会让我们的项目走下坡路，导致设计越来越差、维护工作顾此失彼。的确，大多数专业人士都说代码克隆是毋庸置疑的坏事。但有多坏呢？我们又如何来确定呢？

所以我们可以停下来思考一下，也许可以再问问自己：代码复制到底有多大危害？从长远来看，代码复制是否会带来明显的维护问题？我们到底为什么要复制代码？每个开发人员复制代码的原因是否相同，是否有些原因相对来说更站得住脚？代码复制有无任何可取之处？我们能不能用好代码复制，如果可以的话需要做哪些付出？

这些都是值得研究的好问题。作为软件工程的研究员，我们发现每次在参加关于代码复制的讨论的时候，讨论的前提都是：代码复制是个彻头彻尾的坏事情。但我们的个人经历又让我们坚信其实代码复制并没那么坏。所以我们决定用目前最先进的代码克隆检测工具，对一个大型的开源项目做一些定性和定量研究——结果令我们大吃一惊。

28.1 代码克隆的示例

在我们继续之前，不妨简单地探讨一下代码克隆的问题空间。为此，我们先来看看下面两个函数（均来自Gnumeric 1.6.3版的源代码）：

```
// 两个函数都属于文件 py-gnyneruc.c，版本1.6.3
static PyObject *
py_new_Range_object (GnmRange const *range) {
        py_Range_object *self;
```

```
        self = PyObject_NEW (py_Range_object, &py_Range_object_type);
        if (self == NULL) {
              return NULL;
        }
        self->range = *range;
        return (PyObject *) self;
}

static PyObject *
py_new_RangeRef_object (const GnmRangeRef *range) {
        py_RangeRef_object *self;
        self = PyObject_NEW (py_RangeRef_object, &py_RangeRef_object_type);
        if (self == NULL) {
                return NULL;
        }
        self->range_ref = *range_ref;
        return (PyObject *) self;
}
```

乍一看，好像很明显其中一个函数是从另外一个复制粘贴而来（或者两者都是从别处粘贴过来），因为两者几乎每个语言标记（token）都能完全对上。事实上，稍稍调查一下，就会发现，这些实际上是实例化Python对象的时候的样板代码。这些函数有两点不同，一个很明显，而另一个稍稍微妙一些。比较明显的区别是部分标识符有所不同：第一个函数中的标识符的后缀是`Range`，而第二个是`RangeRef`。稍微微妙一些的区别在参数上：第一个函数的参数是指向对象（这里是C结构体即`struct`）的常量指针，而第二个函数的参数是指向常量对象的指针。后者是编程时候的常用方法，当一个对象被传来传去的时候，我们希望能确定它不会被谁修改。而不同的是，前者所做的事情是保证指针参数不被修改，但是却不能保证指针所指向的对象不被改变。对于有经验的C程序员来说，这更像是一个错误，因为参数的改变不会影响到调用的环境，所以把参数作为常量的意义不大。也许参数申明中的`const`放错位置了。

那么到底是怎么回事？我们的猜测是，第一个函数是原版，而第二个是克隆，因为我们认为开发人员更可能去添加而不是删除一个后缀。此外，我们怀疑在克隆版中这个参数的bug已经被发现并被修复了，但是没有对应地修改原版。如果事实却是如此，那么这就是我们所谓的*维护工作顾此失彼*：在有些版本中bug被解决了，而有些又没有解决。这是我们认为代码克隆最大的毛病。

另一个反对使用代码克隆的主要原因是因为它会让系统设计越来越差。这就有点尴尬了，明明是先有累赘的设计，然后才导致了代码的琐碎而让人搞不明白设计的本意，不是吗？如果我们能把这些重复的东西剥离出来并统一地放到代码库中专门的位置，然后再通过代入不同的参数来使用它们，不是更好吗？用面向对象的术语来说，就是如果我们能将继承和泛型结合来使设计更加简单有效，不是更好吗？

就前一个例子来说，如果对整个代码库没有很详细的了解，就很难知道“正确”的设计应该是什么样的。老方法是有点尴尬，但是代入参数的新方法会好一些吗？过度的剥离会不会导致代码难以读懂，然后又导致维护问题呢？这些也是很好的问题。

28.2　寻找软件中的克隆代码

寻找克隆代码从技术上来说是很有意思的：如果某段代码被复制然后按照新的需求修改了一下，你如何用自动化的工具来识别它呢？如果所有的克隆代码都是一成不变地完全复制粘贴的话（有一些确实如此），那么要找出克隆代码就很简单了。但是，通常粘贴过来的代码都会按照新的需求进行一定修改，比如修改或者删除几行代码或者添加一些新的代码等。所以，也许我们在继续之前应该先搞清楚一个问题：到底什么是克隆代码？

当然，首先我们应该注意一点，那就是几乎所有的克隆代码的探测方法实际上都是在测量代码块的相似度。也就是说，我们通常无法获取实际的记录“Ctrl-C, Ctrl-V”这些事件的日志，我们只能根据代码块的相似度的高低来推断其是否有过复制粘贴。

其次，对于相似度到底代表什么或者相似度多高多低才能算克隆也没有一致的说法。比如这方面的研究员Ira Baxter就喜欢说，“代码克隆就是相似的代码块……基于某个相似度的定义。”检测工具使用的技术各有不同。有些工具把程序看做字符串的序列，并对其进行字面上的比较。有些工具对比标记流（token stream），抽象语法树（abstract syntax trees）以及程序依赖关系图（program dependence graphs）。另一些工具使用度量或者比较程序组件的轻量级语义模型（lightweight semantic models）的方法。还有越来越多的工具开始使用混合的方式，在大范围使用不需要太高计算能力的方法，而在小范围使用需要对计算能力要求较高的方法。

虽然五花八门的技术让这个研究领域显得很有趣，但是实际上在研究中，代码克隆的定义大部分还是得取决于使用的工具。例如，用简单的字符串比较是找不出标识符的变化的。这样就意味着，对于两段代码是不是克隆，不同的工具就有可能得出不同的结果。甚至当专家们坐在同一个房间并亲自对代码进行检查的时候，也常常在“到底多相似才能算克隆”这个问题上达不成共识 [6] 。不用说，共识的缺乏会妨碍研究的进展：由于没有“黄金标准”或者公认的衡量法，很难知道我们的结果到底是好是坏，也很难对于各种工具做横向比较。

即便如此，对于代码克隆的问题域的澄清其实也还是有一些进展的。Bellon等人创立了一套为代码克隆分类的制度，现在已经被业内人士广泛采用 [2] 。

- 1型

  两段代码在字面上完全等同（可能需要忽略代码注释或者纯用于排版的空格）。

- 2型

  除了标识符以及直接写在代码中的常量值（如21、`"test"`等）可能有不同以外，两段代码在字面上完全等同。

- 3型

  除了满足2型克隆的标准以外，属于本类型的两段代码还可能包含一些不同的“分歧”。

正如我们之前所说的，要检测1型克隆非常简单。在用一个简单的词法处理（lexical processing）将代码文本正规化之后，有很多算法都可以对处理好的代码行进行序列分析。

而检测2型克隆就要难多了。词法分析器可以做一些简单的重写（比如，将所有标识符换成“id”，所有的字符串常量都换成“foo”），接下来，要么可以把生成的标记流作为普通的纯文本来

分析，要么可以使用编程语言特定的分析技术来分析。

最有趣的是3型，要找出它们要更难一些，因为他们需要合理的界定值：两段代码必须连续有多少相同才算匹配？两段匹配的代码最多可以有多少“分歧”？如果界定值太宽松的话，随便哪两段代码都可以被认作是3型克隆！

而且并不是所有克隆检测工具都可以整齐地对应到Bellon等人的分类定义。比如有一些基于度量的检测法就不只是纯粹基于程序结构，而是将程序片段的结构和意义综合成为一个个的数字，然后再对比这些数字。

因此，至少在目前这个时刻，我们可以理直气壮地说，对于“我的软件项目中有多少代码克隆”这样的问题，确实没有一个权威的答案。但这并不意味着我们就该洗洗睡了。过去的经验让我们深信，我们能找出很大一部分存在于软件项目中的代码克隆。所以我们决定，使用最先进的检测技术来进行“地毯”式搜索，看看到底结果会怎样。

28.3　对代码克隆行为的调查

在软件工程师之间有一个不成文的规矩，而且大部分人都欣然接受这个规矩，那就是：代码克隆是件坏事。彻头彻尾的坏事。好吧，短期来看克隆确实可以帮你省点事儿，但是长期来说，克隆是件坏事。事实上，Kent Beck在《重构：改善既有代码的设计》[①]（与Martin Fowler同著）一书里 “代码的坏味道”一章中，正好这样写道：

> 坏味道行列中首当其冲的就是重复代码。如果你在一个以上的地点看到相同的程序结构，那么可以肯定：设法将它们合而为一，程序会变得更好。

根据我们的经验和直觉来看，这个观点把复杂的问题给过分简单化了。例如，我们的同事Jim Cordy提醒我们，从软件工程的角度来看，使用已经存在的解决方案有时是必要的：比如像FORTRAN和COBOL这样的语言，语法奇特，进行高度抽象能力也有限，已经存在的解决方案常被视为是可以重用的工具，可以在经过调整之后适用于新的情况。（这听起来好像是应该由软件库来做的事情，但是通常来说这类解决方案无法整齐地打包成库。）

所以我们决定从另一个角度来看这个问题：在商用软件系统中，克隆代码有什么特点？有些什么模式？我们能使用静态分析来识别克隆代码吗？我们能判断什么时候克隆是合理甚至有利的设计方法吗？有了相关知识和实证研究的帮助，我们能不能把代码克隆作为正当的开发方法？

基于同事以及我们自己以前对于代码克隆的检测和分析的探索，我们开始着手创建一个克隆模式的目录。在记录这些模式的时候，我们使用了下面这个模板：

- 模式的名字；
- 主观地描述一下程序员用这个模式来克隆代码的初衷；
- 优点和缺点列表；
- 描述程序员接下来可能面对的管理和长期维护问题；

① 该书中文版已由人民邮电出版社出版。——编者注

- 用代码描述一下这个克隆模式结构上的特征（即如何用工具进行自动识别）；
- 一组已知的案例。

最后我们确定三个截然不同的克隆大类：分叉、模板和定制。在每个大类下面，我们又识别出了很多克隆模式。下面我们将概述一下这些类别和模式，如果需要更详细的信息，你可以阅读我们关于这个主题的论文[5]。

28.3.1 分叉

分叉（forking）是指开发人员想要用已有的解决方案来快速解决另一个环境中的相似问题。一般来说，原版的代码会直接复制到新的源文件，然后原版和克隆版两个版本将被独立地维护，当然可能中间还包含开发团队之间的一些协调工作。分叉有助于保护系统的稳定性，因为可以使变体向边缘发展，远离核心组件。通过这种方式，硬件或者平台变体和系统的其他部分实际上是通过一个虚拟层来互动，即使需要测试尚不成熟的功能，也不会对主系统造成影响。当变体需要独立发展，尤其是其长期发展路线尚不明朗的时候，分叉克隆不失为一个好的办法。

分叉之下，我们发现了以下三种模式。

- 硬件变体

  硬件变体是指：新的硬件和之前的硬件相互之间在程序语义上非常相似，但是又有所不同，所以需要做一些特别的处理。例如，在Linux操作系统中，一个SCSI卡的驱动程序有可能需要支持多个硬件版本。当新的大同小异的SCSI卡发布的时候，新的驱动就有可能基于已有驱动（即基于复制过来的代码）来做。由于驱动程序都是独立维护，所以通常来说新的代码和旧的正常工作的代码会分隔开来，以免产生向后兼容性的问题。此外，驱动程序通常是打包成为单一的程序，想要将通用的小部件重构成可以共享的组件比较困难。

- 平台变体

  平台变体和硬件变体紧密相连。当系统中有一个对应不同功能的抽象层，并且可以针对不同目标平台（或操作系统）来实现的时候，就可能产生平台变体。Apache Portable Runtime（APR）就是一个很好的例子：程序使用由APR提供的通用API来访问系统的底层服务，比如内存分配或者文件I/O。通用API在不同系统上的实现是不同的，但是从代码的层面上来看，常常又是非常相似的。大部分APR实现被划分到各个OS特定的目录，并且各个版本之间的代码克隆度非常高。这样的克隆反而是有好处的，因为开发人员可以对各个变体进行单独维护，并且可以再针对某个OS做特定的修改。不过需要注意的一点是，在针对不同操作系统的APR的开发人员之间有着大量的沟通，从而能够最大限度地减少设计偏差和顾此失彼的维护所带来的风险。

- 实验变体

  采用实验变体方法的时候，通常系统已有一个稳定而且有着很多用户的分支版（branch），但是开发人员想要尝试一些新功能，而这些功能又可能对系统的性能或者可靠性带来一些负面影响。开发人员会创建一个安全的沙盒，用一个克隆系统来实验新的想法和功

能。然后他们会把感觉不错的新功能及时地再移植回主代码库中，而克隆的系统有可能被扔掉，也有可能自己变成新的主代码库。Apache项目、Linux内核项目以及其他很多著名的开源项目都大量地使用了实验变体这种方法。

28.3.2 模板

模板是指在所需的抽象机制（如继承或泛型）不存在的时候，对现有代码进行直接复制的行为。当原版和克隆版的代码需求类似（比如行为需求类似或者都使用某个库）的时候，我们就把它们归为模板一类。当这些需求变化的时候，所有版本必须保持一致。例如下面这段来自Gnumeric 1.6.3版的代码：

```
gnumeric_oct2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
        return val_to_base(ei, argv[0], argv[1],
                8, 2,
                0, GNM_const(7777777777.0),
                V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}

gnumeric_hex2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
         return val_to_base(ei, argv[0], argv[1],
                  16, 2,
                  0, GNM_const(9999999999.0),
                  V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}
```

这些函数的区别只在于函数名和所用的数字常量值。有经验的C程序员可以看出，这是简单的格式转换函数。在面向对象的编程语言中，继承和泛型的存在大大简化了类似功能的实现。不过话说回来，像C这样的流程式的编程语言确实常常需要创造一大堆类似的函数。这个现象通常会造成很多小的代码克隆：如果有`m`种输入格式和`n`种输出格式，那么就必须创建和维护`m*n`个几乎完全相同的函数。虽然这个方法从数学层面上来看确实很笨拙，而且实施起来也很麻烦，但编程语言在表达能力上的限制又使得我们不得不采用它。在这类情况下，代码克隆就无法避免了。在模板类别下，我们发现了4种模式。

- 参数化的代码（Parameterized Code）

  这种模式的特点是克隆代码之间直白而精确的对应（见前面一段代码）。如果有更强大而且表达能力更好的程序语言，只需使用集成或者泛型这样的功能，就可以很容易地把这些克隆合并成一个单独的函数。有时候参数化的代码是由特殊工具自动生成的，以避免意外的错误。在Java还没有加入泛型功能的日子里，JDK曾经用过这种技术。

- 样板文件

  这种模式和代入参数的代码差不多，但是要求要宽松一些，不需要像后者那么精确的对应。这个模式在由旧式流程式程序语言（如COBOL及FORTRAN）写成的系统中尤其常见。在这些语言写成的系统中，已有的代码解决方案被看作是智慧的结晶，值得用来学习、重用

和效仿。在程序语言中缺乏对用户自定义抽象概念的支持往往意味着一个相对简单的问题可能需要冗长、复杂和不便使用的方式来解决。当开发人员第二次遇到类似问题的时候，他就可能选择复制之前的解决方案再进行调整，而不是全部重写，以免遇到过的错误再次发生。样板文件和代入参数的代码的区别就在于，克隆版本之间的差别无法简单地用一个解决方案来概括。

- API协议

  这个模式是样板文件的一个变体。在这个模式中，程序员在调用一个库或者框架的时候，采用了一套推荐的使用规则。比如说，当使用Java SWING API来创建一个按钮的时候，常见的顺序是：创建按钮、添加到容器，再指定动作监听器。同样地，在Unix环境下用C语言来构建一个网络套接字的时候，也需要调用某个库中的一系列既定的函数。而这些库（尤其是框架）的文档通常是以“上手指南”的方式来写的，其中包含了很多代码范例，用来演示各种常见任务的实现方法。这些文档通常鼓励读者复制这些范例到自己的代码，并进行修改以适应自己的需求。

- 编程惯例

  这个模式指的是，在整个代码库中系统地使用程序语言特定的惯例（programming idioms）来执行某些底层任务。例如，在Apache的代码库中，对于在内存池中如何创建一个指向特定平台的数据结构的指针，就有着明确的使用惯例。首先，检查在内存池中的数据结构是否包含指针。如果不存在，那么就给其分配空间，并将其地址赋值给指针。这个惯例的存在是因为APR库使用了类似定义的数据结构，用于指向特定平台的结构（如pthreads）。这些数据结构还将储存和其概念相关的特定平台的数据，如线程的退出状态。此外，这个惯例还有一些小的变体，比如有一些会包含检查内存池是否存在，如果不存在就返回错误。我们发现在APR子系统中，这个惯例出现了至少15次。

28.3.3　定制

当现有代码可以解决一个和当前问题很相似的问题，但又没有相似到可以直接复制代码的时候，就会出现*定制*的情况。有时候定制并不只因为技术原因，而更多是基于管理上的压力，比如代码的所有权问题或者想要把没测试的代码和测试好的代码区分开来等。在这种情况下，想要实现新的功能不能“在本地”修改现有的代码，所以只能将其复制到系统的其他位置进行调整。

定制和分叉、模板有下面一些不同。在定制大类下，新的设计是从原版代码中随着时间推移慢慢衍生出来的，而克隆代码只是实现新设计的起点。从长远来看，克隆的代码和原版的代码之间几乎没有什么协调和同步的工作。在分叉大类下，虽然变体在细节上都是独立发展，但是当共同的外部需求变化的时候，各个变体的开发组之间还是会协商解决问题。在模板大类下，各个代码克隆之间的关系更为紧密，因为正是由于语言的表达能力不够或者系统的设计不好导致了这些克隆不能被合并为一个单独的抽象概念。模板和分叉的目标通常都是高度维持原版的行为，而定制只是为了*重用*原版的行为，通常不需要各个版本的行为一直维持非常相似的状态。像定制这样

没有任何规矩可言的克隆和其他的克隆相比：克隆之间的区别可能更不明显，行为的变更所带来的影响可能更难理解，而对克隆代码的检测也可能更加困难。

在定制类别下，我们发现了两种模式。

- 解决bug的方案

  在开发人员发现了一个bug，但是却没有权限进行修改的时候，就会出现这种模式的克隆。他无权修改原版代码，所以只能把出问题的函数复制粘贴到一个他可以修改的模块或者类里面（而且还可能是在一个高手的指导下这样做的）。在面向对象的程序语言里，这个问题很好解决：你可以创建一个新的继承自有问题的类，并替换出问题的函数。

- 复制并专门化

  当开发人员发现了一个功能，然后想修改这个功能并应用到系统的其他地方的时候，常会使用这个模式。这种复制有可能是小规模和小范围的复制，也有可能是大规模和大范围的复制。LaToza等注意到在微软中，大规模和大范围的复制是一种现行的做法，他们称为“克隆并认领”（clone and own）[8]。当一个开发组希望使用处于其他开发组控制下的特定的功能代码的时候，他们可以选择复制一份原代码到自己的系统中，并按需求修改。但如果要这样做的话，团队之间需要一种默契，那就是克隆代码的团队将对这段代码将来的维护和发展负责，即他们“认领”了这个克隆（更多关于这个主题的信息请参见参考文献[1]和参考文献[4]）。

28.4 我们的研究

我们提出的分类是基于我们对于多个开源软件系统代码克隆的分析经验。这些系统包括Linux内核、Apache httpd网页服务器、Postgre SQL关系数据库系统、Gnumeric电子表格程序、Columba电子邮件客户端、九个文本编辑器（包括vim和emacs）以及八个X11窗口管理器。对于每一个模式，我们都在多个系统中有多次发现，所以我们非常确信这些模式是真实而且普遍存在的，而不是特定于某个系统或者某个类型的应用程序。虽然我们非常确信代码克隆常被作为一种正当的开发方式，但是我们却缺乏具体可量化的证据。所以我们研究了两个不同领域的大型开源系统，并尝试去测量：（至少在这两个系统中）代码克隆作为一个正当的设计工具的应用有多普遍。

我们使用了我们自己的叫做CLICS（CLoning Interpretation and Categorization System，克隆解读及分类系统）的克隆检测工具来对这些大型系统进行遍历和分类。CLICS将源代码标记化，并用后缀数组来寻找带参数的字符串匹配。这和CCfinder等工具使用的方法类似。为了检测那些修改了变量名字的克隆，这个工具将所有标识符都统一替换成一个替代标记，也即是说，任意两个标识符都可以相互匹配。而剩下的标记，如关键字、操作符和分隔符等，都在识别克隆代码段的过程中起重要作用。不过，这个方法将会导致大量的误报，因为忽略实际标识符使得很多有用的信息丢失。因此，我们对初步结果进行了严格地筛选，以去除可能是误报的结果。例如，在经过我们这样的处理之后，在C语言及类似的语言中的所有`switch`语句看上去都很像，即便它们实际做的事情大相径庭。对于这些特别容易引起误报的情况，我们使用了更为严格的筛选条件。

如果两段代码通过了我们的匹配条件以及自动化过滤，我们就认为他们是一对克隆。然后我

们观察了代码所在的源文件区域，以更好地理解这些结果。我们采取的详细步骤可以在我们的论文中找到，而简化版的步骤描述就是：我们将文件分割成一个个的函数界限以及其他的“区域”（例如struct定义区域等），再认定代码区域之间的克隆关系。我们找到克隆代码所在的区域，再寻找和这个区域的代码有克隆关系的其他区域。我们管这些区域叫做克隆区域组（Regional Groups of Clones，RGC）。这些克隆区域组代表了软件系统中代码克隆的粗略情况。在这个层面上的情况可能对开发人员更有意义。我们研究的大部分使用克隆区域组作为基础，这是因为在我们看来，用克隆区域组可以比较直观地看出代码克隆是否重要。所以对于每个系统，我们的报告都包括找到的克隆代码对以及克隆区域组的数量。

对于我们的研究，我们决定只当有30～60个连续匹配的标记的时候，才考虑两段代码存在克隆的可能性。这两个数字是根据我们此前在这个领域的工作经验得出的。

我们挑选了曾经研究过的两个大型开源系统：Apache和Gnumeric。我们选择这两个系统的原因是：它们都很成功、资格很老、规模相似并且来自不同的领域。在版本上，我们选择了Apache的2.2.4版，其中共有300 000行代码分布于783个文件之中，以及Gnumeric的1.6.3版共有300 000行代码分布于530个文件中。

使用我们之前描述的规则，CLICS在Apache项目中找出了21 270个克隆和1580个克隆区域组，在Gnumeric项目中找出了11 400个克隆和3437个克隆区域组。看上去挺多的，不是么？多是很多，但是对于我们分析的这类系统来说并不意外。也就是说，代码克隆在大型系统中很常见，甚至可能比你想象的要常见得多。

我们分别从这两个系统中随机选择了100个克隆区域组。然后，我们人工检查了每一个组，并试着回答：这个克隆是该用，不该用或者干脆就是根本无法避免。我们认为该用的克隆是指那些对其他可能的解决方案有一定改进作用的克隆。不该用的克隆是指那些明显可以有更好的替代解决方案的克隆，也指那些可能是由于开发人员偷懒、设计偏差或者其他不良情况造成的克隆。无法避免的克隆是指那些由于过于琐碎而难以做重构所造成的克隆，也指那些没有其他合理的替代解决方案而造成的非用克隆不可的情况。最常被视为无法避免的克隆是API协议，因为这些协议难以被抽象化，而且开发人员通常对原版代码也没有控制权。我们也发现了一些误报的情况：Apache的克隆区域组中有7%是误报，而在Gnumeric的克隆区域组中，我们通过人工判断，有29%是误报。这些误报造成的结果就是在下面这些表格中，每个系统的总和不是100。

下面这个表格总结了我们的发现。我们只显示了连续匹配60个标记的克隆，因为匹配长一些的克隆更值得我们关注，而且也不太可能是误报。经我们鉴定，总的来说有大概有35.4%的克隆是该用的，而不该用的稍微多一些，还有15.2%的克隆是完全无法避免的。对我们来说，这很好地证明了开源软件开发人员常常用代码克隆作为正当的开发工具。

分类	模　式	Apache：该用	Apache：无法避免	Apache：不该用	Gnumeric：该用	Gnumeric：无法避免	Gnumeric：不该用
分叉	硬件变体	0	0	0	0	0	0
分叉	平台变体	10	0	0	0	0	0

（续）

分类	模　　式	Apache：该用	Apache：无法避免	Apache：不该用	Gnumeric：该用	Gnumeric：无法避免	Gnumeric：不该用
分叉	实验变体	4	0	0	0	0	0
模板	样板代码	5	0	0	6	0	1
模板	编程惯例	0	0	12	1	0	0
模板	API协议	0	17	0	0	8	1
模板	参数化的代码	5	1	12	10	0	24
定制	bug解决方案	0	0	0	0	0	0
定制	复制并专门化	12	0	4	15	0	1
其他		3	0	8	1	0	3
总共		39	18	36	33	8	30

必须要说明的是，我们这绝对不是在对克隆的相对危害性下结论。我们尽了最大的努力来完善我们的克隆分类，但是肯定也会有其他的分类法，甚至可能有相关的组织性原则①（organizing principles）的存在。除此之外，任何实证研究本身也不可能做到完美。首先，我们自行设计和执行了这次研究。如果能启用更为中立的第三方人员来判断克隆的危害的话，就可以降低偏见的问题，不过这就需要牺牲部分的专业判断能力。而且，我们只研究了两个系统，而整个大千世界中有数不清的程序种类。从统计学的角度来看，只有两个系统既无法证明统计结果有足够的显著性，也无法证明其具备代表性。而且两个系统都是开源的，这就有可能使得我们的结果更加片面。最后，对于有一些问题，我们并没有可以作为答案的数据，比如克隆对于代码质量的长期影响，顾此失彼的维护带来的风险是否值得注意等，不过已经现在已经开始有一些相关的研究了[7]。我们认为，代码克隆作为正当的开发方法得到了强有力的证据支持。

虽然详细的研究结果可以在我们的论文中找到，但在这里值得指出的是，我们在两个系统中找到的一些有趣的区别。其中之一是，我们发现Apache大量地（并且我们认为是合理地）使用了分叉克隆，而Gnumeric没有。这并不令我们意外，因为Apache提供了一大套很底层的服务。这些服务的目的是直接在各类平台之上运行，使得APR所提供的虚拟环境（我们已知道APR使用了很多平台变体模式）可以发挥作用。而Gnumeric的跨平台性依赖于GTK控件集，而这个控件集并非Gnumeric的一部分。我们还发现Gnumeric相对于Apache似乎使用了更多的参数化的代码模式。经过人工检查我们发现，很多基于用户界面操作的功能的实现方式都非常相似，所以造成了这种克隆模式。最后，需要注意的是，这些数字只是基于对来自两个系统的克隆区域组的100个随机取样的结果，分别占了Apache克隆区域组总数的6%，Gnumeric的3%。

28.5 总结

我们开篇问了一个简单的问题：代码克隆是不是真的像有些人说的那么一无是处？我们在本

① 指用于帮助分类的基准。——译者注

章中对大型开源软件中代码克隆的研究结果回答了这个问题：代码克隆非但不是一无是处，而且在有些时候反而是正确而正当的事情！我们通过思考实验[①]（thought experiement）得出了这个假说，并得到了实证研究的支持，所以现在我们非常确定我们的结论是正确的。当然，有趣的研究通常会在回答问题的时候，也带来很多新的问题，我们的研究也不例外。在回答了开篇的问题之后，还有这样一些问题值得我们关注：除了已经介绍的，另外还有些什么代码克隆模式？各种类型的克隆的风险孰高孰低？在不同的程序领域中，克隆的风险有什么不同？跨应用程序进行代码克隆会有什么后果？有哪些工具或者程序语言支持可以缓解维护的压力并降低克隆的风险？好了，这些都是很好的问题，那就让我们留待下回分解吧。

28.6　参考文献

[1] [Al-Ekram 2005] Al-Ekram, R., C.J. Kapser, R.C. Holt, and M.W. Godfrey. 2005. Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems. *Proceedings of the 2005 International Symposium on Empirical Software Engineering* (ISESE-05).

[2] [Bellon 2007] Bellon, S., R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9): 577-591.

[3] [Cordy 2003] Cordy, J.R. 2003. Comprehending reality—Practical barriers to industrial adoption of software maintenance automation. *Proceedings of the 11th IEEE International Workshop on Program Comprehension*: 196-206.

[4] [German 2009] German D.M., M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol. 2009. Code siblings: Technical and legal implications of copying code between applications. *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*: 81-90.

[5] [Kapser and Godfrey 2008] Kapser, C.J., and M.W. Godfrey. 2008. “Cloning considered harmful”considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6).

[6] [Kapser et al. 2007] Kapser, C.J., P. Anderson, M.W. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, and P. Wei β gerber. 2007. Subjectivity in clone judgment: Can we ever agree? *Proceedings of Dagstuhl Seminar #06301*.

[7] [Krinke 2007] Krinke, J. 2007. A study of consistent and inconsistent changes to code clones. *Proceedings of the 14th Working Conference on Reverse Engineering*.

[8] [LaToza et al. 2006] LaToza, T., G. Venolia, and R. DeLine. 2006. Maintaining mental models: A study of developer work habits. *Proceedings of the 28th International Conference on Software Engineering:* 492-501.

① 思考实验指在脑海中（而非现实中）进行的实验。——译者注

第29章

你的API有多好用

Steven Clarke

1999年11月，我加入了Visual Studio（下称“VS”）的用户体验团队进行用户体验方面的研究。我很高兴成为这个团队的一员，因为我在前一份工作中使用的产品就是他们设计的。还是在摩托罗拉工作的时候，我就总是想要对VS的用户体验进行大刀阔斧的改革，现在终于有机会了。虽然我现在已经很明白改善开发人员体验的重点不一定在开发工具上，但当初意识到这个问题却花了我将近一年的时间。

当刚加入这个团队的时候，我将所有注意力都集中在了理解开发人员使用VS的方式上。我和我的同事们一起花了无数个小时观察开发人员如何使用VS，包括他们如何创建新项目、如何添加新的类到已有的C++项目中以及如何调试Web服务等。我们发现了很多有用的信息，可以更好地了解用户，并利用这些了解来设计和创造一个更好的开发工具。

但是好像少了点什么。虽然开发人员对于开发工具的使用体验不断在改善，他们的整体开发体验却并没有提高那么多。所以我们需要改变关注的重点。与其纠结于开发人员很少做的事情（谁也不会天天创建新项目吧），我们应该更多地将注意力放在开发人员经常做的事情上。而所谓常做的事情，就是读、写、调试那些使用了很多不同API的代码。所以我们开始了度量和改善开发人员的类库和API使用体验的一段漫漫长路。

刚开始的时候我们没找对方法。不过，多亏毅力和对于改善用户体验的强烈愿望，我们还是成功地了解了一点，那就是怎样的API才算好用。在这个过程中，我们学到了很多教训，有一些甚至让我们对于开发人员如何写代码这个最基本的事情都产生了疑问。在这一章中，我将和你分享我们一路走来的这些故事。

29.1 为什么研究 API 的易用性很重要

和图形用户界面不同，开发人员并不直接通过可视化操作办法来和API进行互动。正好相反，开发人员需要将文字写入文本编辑器，然后编译和运行程序。所以说开发人员并不直接和API互动，而是直接和他们写代码和运行代码的工具互动。很显然这些工具的易用性对于开发人员编写和运行这些使用API的代码的能力有着很大的影响。

但是只需要知道这个就行了吗？如果有一个理想世界，而这个世界中的开发工具是完美的……那么开发人员编写和运行这些使用API的代码的体验也就完美了吗？让我们假设工具是完美的，再考虑一下设计拙劣的API对于开发人员的影响。

API提供了一些组件来执行一些常见的任务，它们本来应该是用来增加开发人员的效率的。诚然，很多时候开发人员需要使用API来完成一些任务，因为这是唯一能访问封装好的系统功能的办法，而系统功能无法从零开始实现。就算这些系统隐藏功能无需特殊权限就能直接调用，API也可以节省开发人员实现这些功能组件的时间和精力。不幸的是，设计糟糕的API让人无法分辨使用它节省下来的时间能不能抵消它的使用成本。

关于这一点，Alan Blackwell提出了一个模型，用来判断是否应该使用自定义的解决方案而不是现有的API[1]。在这个模型中，开发人员需要估计使用API的认知成本（即知道是什么），以及学习成本（即知道怎么用），然后再用回报（即使用API可能节省的时间和精力）来权衡这些成本。这个模式已经考虑到了没有回报的可能性。

在进行分析时，开发人员将使用API设计中的提示（如类的名称、方法的名称以及相关文档）来估计所需的成本。开发人员可能会认出API所使用的特定设计模式，这将能帮助他更好地学习如何使用API。开发人员也有可能用过API的某些部件，这也可能让人找到学习的方法。此外，开发人员对于API相关领域的了解，也能成为学习API的过程中的另一种视角。API的指示、设计模式、开发人员早前使用API的经验以及对API工作领域的了解，都将帮助他们估计自己实现这样的功能需要的工作量。然后，他便可以决定是否要使用API了。

不过，也有很多事情会妨碍开发人员准确地估计所需工作量。

- 类的名称或者文档可能有误导性或者不准确。这些东西暗示API做的事情和API实际上做的事情可能并不一致。Martin Robillard表示，学习API的最大障碍，就是缺乏描述API和其使用方法的良好资源[9]。
- 关于名称的另一个常见问题是多重含义。要确定具体名称，通常需要首先对API设计的概念模型有所了解。
- 此前使用过的API的部件和剩下的部件的工作方式可能并不一致，尤其是当API的设计理念并没有统一地贯穿于整个系统中的时候。
- 开发人员所预想的API的工作方式和其实际的工作方式可能有很大不同。

遗憾的是，开发人员很难立刻就断定API的设计不一致、类和方法的名称有问题或者和他所预想的工作方式不同。在估计API的使用成本时，有一个很大的问题，那就是开发人员常常需要自己先用这个API做一个小的程序，才能据此估计在更大一些的程序中使用这个API需要花费多少时间。只有当他开始用这个API写代码的时候，他才能发现这个API在设计上的奇特和怪异之处。

最重要的是，这为开发人员已经估计好的工作量加上了额外的负担，使用API的成本很容易就会超过自己写解决方案的成本。正因如此，开发人员可能会觉得API不但不能提升他们的效率，实际上反而降低了他们的效率。

所以致力于帮助开发人员了解API的使用成本的论坛、网站和博客一窝蜂地涌现了出来。比如StackOverflow.com这样的网站，就让开发人员可以提一些关于API使用方面的问题并找到答案。

结果就是开发人员通常搞不明白如何使用API，只得花费大量的时间来浏览论坛、提问题和寻找答案，而不是高效地写代码。这打击了他们的自尊。他们感觉自己做的事情就只是不断地在各种论坛上施展搜索问题的技巧，并从搜索结果中复制粘贴代码。感觉程序员好像并不是在“写”程序了一样。

而且即便是完美的工具也不能解决设计不佳的API的问题。虽然有很多工具可以帮助开发人员编写和运行代码，但是当面对设计拙劣的API的时候，大部分的成本其实是认知成本。开发人员必须思考怎样才能将他们的想法用这些像咒语一样的类和方法表示出来。调试器和其他类似的分析工具可以帮助开发人员更迅速地认识到API的预期工作模式和实际工作模式的区别。但是开发人员仍然需要处理这些工具生成的信息。而且API的工作模式和开发人员的预期差别越大，开发人员就需要越多的思维转换来克服这种差别。

与此形成鲜明对比的是，设计良好的API允许开发人员准确地估计其使用成本。API中使用的类和方法的名称让开发人员可以迅速了解这些类和方法所做的事情。API中的类和方法正好符合开发人员的期望。API的各个部分的类和方法的工作方式都一致。然后开发人员便能进入“兴奋状态”，那些代码就像直接从他们思想中流到屏幕上一样。即使开发人员使用的工具并不是最好的，他们仍然可以很轻松地使用该API，因为和设计得很差的API不同，使用设计良好的API并不需要太依赖工具。在理想世界里，开发工具其实只有输入代码和运行程序两个作用。

29.2 研究 API 易用性的首次尝试

在开发人员的整体开发体验中，API的易用性非常重要，这一点似乎很明显。但是当我刚刚加入VS用户体验团队的时候，我却并没有意识到。

作为一个曾经使用过VS的开发人员来说，我常常被这个工具搞得头大。编译器的设置既难找到，又难设置。配置库的路径显得十分麻烦。这些就是在我刚刚加入VS用户体验团队的时候想要解决的易用性问题。

这并不是说我就没有发现过微软API的问题。正好相反，我遇到的最大的困难中，有一些正好是和学习微软基础类（MFC）的工作方式相关的。不过我没像平时一样抱怨工具不好用，实际上当这些MFC不像我想象的那样工作的时候，我往往是责怪自己学习能力不够。我从没想过这个问题可能和API的设计有关。

对于设计拙劣的API来说，人们的这种反应非常普遍。很多开发人员都不肯面对现实，认为API难用的原因是他们使用API的经验还不够。只要有足够的时间，他们就能搞明白。我们已经听过很多开发人员把问题往自己身上揽了，说他们不够聪明所以不会用某个API。

所以，虽然我是VS用户体验团队的一部分，但是我没能意识到API的易用性是值得研究的。这个想法来自于我此前工作的团队，当时我们正在开发一套全新的API，称之为ATL Server。

ATL Server是一套用于开发Web应用程序的C++ API。很多ATL Server的开发人员、测试人员以及项目经理都曾经开发过MFC API。他们在开发MFC时学到的经验是，开发人员有时候并不知道怎样设计好API。他们在开发的时候就已经知道有一些设计上的决定会带来易用性的问题。但

是现在MFC API已经发布而且很多用户已经在使用了，要做大的改变就太迟了，因为任何大的改变都有可能使用户的现有代码停止工作。所以这个团队开始开发一套全新的API，这让他们可以在正式发布之前就确定他们的设计是否正确。

由于我是为这个团队指派的易用性工程师，他们就向我寻求帮助。然后我便开始尝试找出研究API易用性的方法。

我首先想到的是：API有那么多种用法，我怎么才能把它们都研究一遍呢？API能做的事情实在是太多了，我不可能一一研究。所以，我将研究的重点放在了API的整体结构上。因为我认为弄清楚API中类的组织结构是否对于开发人员来说有意义非常重要。因为这也将有助于解决与API的规模相关的问题，并让我可以测量开发人员对于ATL Server的理解和其实际实现的相似度。我的想法是，只要能找出开发人员的理解和ATL Server的实际实现的不同之处，就能帮助开发团队做出改进。

为此，我用卡片分类法来分析了该API。在卡片分类法中，每个参与者都会拿到一叠卡片。每张卡片上都会有一个名字（或者一个概念），用来代表某个对象，如网页、产品或者方法等。参与者需要一张一张地将他们认为最相近的卡片放到一堆。参与者可以根据自己的判断标准将这些卡片分类。当所有的参与者都完成卡片分类之后，研究人员将对这些分类进行聚类分析，找出哪些卡片更像是一类的。分析的结果（即每张卡片最常见的分类）用树状图来表示。卡片分类法应该是研究开发人员的概念模型最合适的方法了。

29.2.1　研究的设计

共有14名经验丰富的C++开发人员参与了这次研究。每个人开发人员每周都有超过20个小时需要用C++来进行开发，工具是VC++ v6.0（研究是在2000年3月进行的）。这些开发人员都没有太多Web相关的开发经验。由于ATL Server的重要设计目标就是让没有什么Web开发经验的开发人员也能顺利进行开发，所以这对这次研究来说正好。

我从ATL Server的API中找出了24个核心类，作为本次研究的重点。表29-1列出了这些类。

表29-1　ATL Server的核心类

CStencil	CMultiPartFormParser	CWriteStreamHelper	CHtmlStencil
CISAPIExtension	CHttpRequestParams	CValidateObject	CStencilCache
CValidateContext	CHttpRequestFile	CDBSessionDataSource	CHttpResponse
TServerContext	CPersistSessionDB	CHttpRequest	CCookie
CSessionNameGenerator	CDefaultErrorProvider	CHtmlTagreplacer	CSessionStateService
CSessionState	CRequestHandlerT	CPersistSessionMem	CDllCache

团队的技术文档专家为这些类的各个功能都撰写了短小简明的介绍。这些介绍基本上都是一两句话长。每个介绍都被印到了一张索引卡片上。例如，用于设置/提取类的成员变量的赋值/访问方法对应的就是写着“变量管理”的卡片。所有类都包含的方法（如构造、解析方法）没有列出。

这些类所实现的功能共汇成了59张卡片。

这些卡片只有功能的介绍，没有类的名字。

以`CHttpRequest`类为例子，它包含了四个主要的功能。这些功能被分别写到四张卡片上：

- 获取HTTP请求数据（如名-值对、表单字段）；
- 用当前请使用求的信息来初始化对象；
- 维护请求的缓存；
- 提取请求和服务器变量。

最后，每个参与者还将拿到一份简短的ATL Server的概览文档作为参考。

参与者共有90分钟的时间来分析理解这个框架，并将59张卡片按照他们的理解分类或者分组。参与者归纳的分类数量最少的有8个，最多的有14个。我们用电子表格记录下这些分类，并对这些数据进行了聚类分析。

29.2.2 第一次研究的结论摘要

聚类研究表明，这些分类有三个常见的类群。第一个类群包含了2个分类，第二个包含了8个，第三个包含了其余的（除开两个不属于任何类群的）。

这些类群之间几乎没有什么共同点。研究小组花费了大量的时间，想要确定到底这样的类群分布代表着什么，但是毫无收获。因为聚类分析的结果是参与者分为三个完全不同的类群，我们只能认为这些参与者们的概念模型没有任何共通性。

看来第一次实验的结果让我们失望了。三个类群完全没有任何意义，也不能用来确定改善的措施。看上去似乎花在设计、执行和分析研究上的时间都浪费了。

但是从长远来看，这项研究反而从完全不同的方面给我们上了重要的一课。它使我们知道：API的易用性并不是整体API的一项功能。只靠研究API的设计架构无法知道所有的易用性相关问题。

我一直执着于用卡片分类法进行研究，因为我一直担心规模的问题：我怎么才能把一个类里面的所有功能的所有用法都考虑进来呢？不过在进行了这次研究之后，我终于发现原来我的方向错了。在实际使用中，很少有开发人员会去关注API的整体结构问题。他们一般只关心他们需要用API来执行的任务。

不同的人会对API有不同的使用感受，因为第一他们需要达到的目的不同，第二他们有着不同的背景，第三他们有着不同的工作方式。一味想去确定API的易用性，而不把这些因素都考虑进来的话，就容易像我做的那个卡片分类法那样，得出无效的结果。

即便卡片分类法能找出共同的类群，我们也无法弄明白它们代表什么。如果API已经按照分析出来的功能类群进行了修改，那么我们又该如何来确定这种修改对于API的易用性有多大影响呢？我们只知道API和参与者的概念模型对应起来了，但我们能确定这是件好事吗？我们真的应该将API的设计和还没有用过这些API的开发人员的概念模型对应起来吗？

实际上，这些正是我在设计和执行卡片分类研究时想要解答的问题，但是事后再来回顾时，我却发现这些问题其实问错了。卡片分类研究的结果使我认识到，如果想要真正地了解API的易用性，就必须研究开发人员如何在实际工作中使用这些API来解决实际问题的。

幸运的是微软的企业文化帮了忙，即便我此前的卡片分类研究并没有得出任何有用的结果，我仍被鼓励继续尝试测量和描述API易用性的方法。

29.3　如果一开始你没有成功

第二次研究API易用性是在18个月以后。那时，我正与VS用户体验团队的同事们一起进行一项研究，即经验丰富的Visual Basic 6（下称“VB6”）开发人员能不能用Visual Basic.NET（下称“VB.NET”）来写基于.NET框架的应用程序。

VB.NET是Visual Basic程序语言的最新版本，和此前的版本（如VB6）有着本质上的不同。它们最大的区别是：VB.NET的框架提供了和VB6类似的功能，但却是由一套完全不同的类和命名空间组成。此外还有如下一些区别。例如，在VB.NET中，默认数组索引的起始值不能从0改成1，而VB6能。

我们的团队想看看初次接触VB.NET的开发人员在用它来执行一些简单任务（如从硬盘上读写文本文件）时的体验。有了此前卡片分类法的教训，我们决定换一种方法，即让参与者在我们的易用性实验室中用VB.NET来执行一些特定的任务。我们让这些参与者在执行这些任务的时候大声把自己的想法说出来，并把每个参与者执行任务的过程都录下来。我们将对每个参与者的录像进行分析，找出他们在行为上的共同点和有趣的亮点，以及任何能帮助我们更好了解VB.NET以及.NET框架的信息。

29.3.1　第二次研究的设计

我们招募了8位开发人员，每个人都称自己每个星期至少会花20个小时在VB6的开发上。我们让他们用VB.NET和.NET框架来写一个程序，这个程序将从硬盘上读写文本文件。每个参与者都有两个小时可以完成这项任务。在写程序的过程中，他们可以随意地浏览所有可用的.NET框架中的类的文档。每个参与者都在我们位于华盛顿州雷德蒙市办公区的易用性实验室中独立完成这些任务。在单面镜背后，有一个易用性工程师将操作摄像机并做笔记。我们的易用性工程师只为参与者解决VS预发布版的bug和程序崩溃问题，除此以外不会从任何其他方面帮助参与者们执行任务。

29.3.2　第二次研究的结论摘要

和上一次做的卡片分类法不同，这次的结果就很有启示意义了。最大的新闻就是没有一位参与者能在两个小时之内完成交给的任务。

这一点让我们非常意外。我们并不指望每个参与者都能顺利完成任务，但是却完全没有意料到结果会是一败涂地。我们的第一个反应是，这些参与者可能并不符合我们招募研究对象的要求，也许他们根本就不是经验丰富的程序员。但是从录像上来看，事实并非如此。录像清楚地显示，每个参与者都是有着丰富经验的程序员，而且对于学习使用新的程序语言和

编程环境并不畏惧。

那怎么还没人成功完成任务呢？我们在研究完成之后对录像进行了详细的分析。我们在分析完了所有的录像并将最有代表性的部分剪辑出来之后，几乎立刻就将这些剪辑好的代表录像发送到了VS开发团队手上。这些录像剪辑展示了参与者们在执行任务的过程中遇到的各种困难。我们对造成这些困难的原因做了一些假设。

第一个假设是：文档出问题了。之所以提出这个假设是因为我们观察到每个参与者都花费了大部分的时间在搜索和查找文档上。他们在文档中寻找那些可以帮助他们从硬盘上读写文件的类或方法。在我们进行此次研究的时候（2002年夏），这些文档还没有加入任何执行常见任务（例如从硬盘上读写文件）的代码范例。

例如，下面这段关于一个参与者如何寻找`System.IO`命名空间的转述记录就很典型。

参与者：

[念出类的名字]

Binary reader。Buffered stream。读和写……

[暂停。把类的列表往下拉]

让我再试试……stream writer。实现一个把字符用指定编码来写到一个文本流的文本写入器。唔……储存基础字符串。字符串生成器。文本……唔。Text reader，text writer。代表一个写入器可以写入按顺序排列的字符序列。这是个抽象类。

[暂停]

唔……

[把类的列表来回拉]

这个，你看吧，我看这个像是，你看看，更像是底层的东西。但我只想创建一个文本文件。呃……

[鼠标指向某一个类的描述]

按编码把字符写入到流。我不确定这个……很明显是一个把字符写入到流的文本写入器。

[点击链接对TextWriter类进行查看。然后查看其派生出来的类]

System.IO.StringWriter。这个看上去好像比较底层，不是我要找的文本写入器，但我也不确定。呃……

[浏览TextWrite类的介绍]

文本写入器应该是为输出字符而设计的，而这个stream类是为了输入输出字节的。

[叹气。点击链接查看TextWriter的成员变量和方法]

可能这个从没人做过吧。

这段对话所对应的视频非常具有说服力，而我们也正好用它来在VS产品团队中宣传我们的研究结果。这些重要的视频资料使得我们对VS的用户体验有了深入的理解，也让所有人都非常渴望找出问题所在以及解决办法。在另一段录像里，参与者正在和编译错误做斗争。

参与者：

好吧。我们又回到这个错误了。

[阅读编译错误讯息]

System……的值。无法将System.IO.Filestream类型的值转换为System.IO.StreamWriter类型的值。搞不懂……

[暂停]

你是在逗我玩儿吗？

很多开发人员揪住一点不放：那就是参与者们大部分的时间都花在了浏览文档上。所以他们觉得，要想改善用户体验，最好的办法就是修改文档。如果文档可以提供一些示例代码或者代码片段，开发人员就可以把这些代码复制粘贴到自己的项目中。

但是，我们想要弄明白这些示例代码的缺失到底是不是实验室中所发现的易用性问题的真正原因。虽然在这个阶段（项目的生命周期已度过四分之三），修改文档也许比修改API的成本要低一些，但是我们也需要考虑万一文档并不是引发易用性问题的真正原因的情况，那样的话文档就白修改了。

为了让我们更有信心地分析这些问题，我们需要一个分析框架或一种语言来作为我们分析的依据。由于产品团队在看了录像之后已经很确信问题在于文档，所以如果没有这套分析框架的话，我们将很难向他们去解释和证明我们的进一步分析。最后我们采用了认知维度（cognitive dimensions）[8]作为框架。

29.3.3　认知维度

“认知维度”是一套用于分析编程语言易用性的框架，还可以推而广之用来分析任意一种表示法，比如API。这套框架包括13个不同的维度，这些维度用于描述以及衡量表示法的易用性相关层面。

这些维度实际上组成了对表示法设计的全面调查，用以测量其易用性的方方面面。对于测量结果我们可以找到一个基准（比如某一类开发人员的偏好）来进行比较，并用这些测量和比较的结果来确定改善措施，即如何使表示法更符合基准人群的需求。

这类分析框架所得出的测量结果并不是一堆数字，甚至不一定是一些绝对的值，而通常是在框架定义的范围内进行评级。虽然这样的做法可能让我们对测量结果的准确度产生怀疑，但是这个框架的魅力正在于此：它识别并提供了相关词汇和定义，使我们可以基于一个标准来细致地讨论表示法的易用性的各个层面。如果没有这样一个框架，任何关于表示法的研究都会陷入困境，因为易用性的特质并没有一个统一的定义和标准，这使得研究人员们对于一些重要问题（比如哪些层面对易用性的影响最大）几乎无法达成共识。

我们对这些维度的名称做了一些修改，使它们更符合我们所研究的API易用性[3]这个范围。对于每个维度的简短描述如下。

- 抽象等级

  API所表现出来的最低及最高的抽象等级，以及目标开发人员所用到的最低及最高抽象等级。

- 认知方式

  API所提出的认知需求，以及目标开发人员可以用到的认知方式 。
- 工作框架

  想有效地使用API所需掌握的知识量（即开发人员的工作集①）。
- 工作的步骤单位

  在一个步骤内可以或者必须完成多少编程任务。
- 渐进式评估

  代码只要完整到什么程度就能执行并将其行为反馈给开发人员。
- 过早的承诺

  开发人员在为了某项任务而编写代码之前，需要做多少决定及这些决定产生的后果。
- 渗透性

  API如何让组件的探知、分析和理解更容易，以及目标开发人员是如何找出所需要的东西的。
- API精细化

  API必须做到何种程度才能符合目标开发人员的需求 。
- API僵化程度

  由API的性质所决定的修改的障碍，以及目标开发人员需要多少工作量才能进行修改。
- 一致性

  在学习了API的一部分之后，能从中推导出多少其他部分的使用方法。
- 角色的表现力

  API的每个组件之间的关系的清晰程度，以及最终程序的整体清晰程度。
- 领域的对应度

  API与其所在领域的对应程度，以及开发人员如果想要完成某项任务所需要了解的特殊技巧。

我们选择这个框架的主要原因是它号称很容易使用。Green和Petre声称，认知维度框架可以被完全没有进行过正式培训或者完全没有认知心理学背景的人使用[8]。我们相信，对于审阅研究结果的开发团队来说，要想读懂这些结果，就必须使用这样一个与之对应的框架（或者语言）作为基础。

我们使用了这个框架来描述参与者执行每项任务的体验。我们审阅了所有的录像，并把每个我们观察到的有趣时刻都剪切了下来。然后我们将每段录像剪辑都用我们认为相关的认知维度做好标记，再按维度来评价各个剪辑。也就是说，我们把所有“抽象等级”的剪辑一起评价了，所有“僵化程度”的剪辑也一起评价了，等等。这种方法使我们能从另一个观点来审视这些录像。我们用这种观点来更清晰地描述我们观察到的难题。

重要的是，我们得以用认知维度框架来说服开发团队：单单改善文档是远远不够的。虽然改

① Working set，指应用程序在某个时间范围内会引用到的内存，此处为比喻。——译者注

善文档是很重要的，但是我们的分析告诉我们，要改善易用性还需要做更多。

认知维度分析表明，最重要最根本的是和API的抽象程度相关的问题。为了更详细地说明这一点，我们可以看看某个参与者要想读取文本文件所需要写的代码：

```
Imports System
Imports System.IO

Class Test
    Public Shared Sub Main()
        Try
            ' Create an instance of StreamReader to read from a file.
            Dim sr As StreamReader = New StreamReader("TestFile.txt")
            Dim line As String
            ' Read and display the lines from the file until the end
            ' of the file is reached.
            Do
                line = sr.ReadLine()
                Console.WriteLine(Line)
            Loop Until line Is Nothing
            sr.Close()
        Catch E As FileNotFoundException
            ' Let the user know what went wrong.
            Console.WriteLine("The file could not be found:")
            Console.WriteLine(E.Message)
        End Try
    End Sub
End Class
```

写入文本文件也是类似：

```
Imports System
Imports System.IO

Class Test
    Public Shared Sub Main()
        ' Create an instance of StreamWriter to write text to a file.
        Dim sw As StreamWriter = New StreamWriter("TestFile.txt")
        ' Add some text to the file.
        sw.Write("This is the ")
        sw.WriteLine("header for the file.")
        sw.WriteLine("-------------------")
        ' Arbitrary objects can also be written to the file.
        sw.Write("The date is: ")
        sw.WriteLine(DateTime.Now)
        sw.Close()
    End Sub
End Class
```

这些代码看上去并不特别复杂。`StreamReader`或者`StreamWriter`的构造函数使用一个表示文件路径的字符串作为参数。然后，它们分别用于执行从硬盘读取数据或者写入数据到硬盘的任务。

如果你已经知道怎么来完成这些任务，然后又看了参与者们如何绞尽脑汁地写这些代码，那

么自然就很容易直接跳到结论，说问题的原因是文档中没有包含执行这种简单任务的示例代码。毕竟这些代码已经简单得不能再简单了嘛。

但是，当我们把这些录像剪辑和相关的参与者的评论一起看的时候，我们就会发现，问题并不在于有没有示例代码。例如，重新阅读此前引用过的一段参与者的解说：

参与者：

System.IO.StringWriter。这个看上去好像比较底层，不是我要找的文本写入器，但我也不确定。呃……

[浏览TextWrite类的介绍]

文本写入器应该是为输出字符而设计的，而这个stream类是为了输入输出字节的。

[叹气。点击链接查看TextWriter的成员变量和方法]

可能这个从没人做过吧。

当参与者说这个看起来太底层了，他指的是他正在浏览的类的抽象程度。相比起现在的API中非常灵活，但又有些抽象的、基于流的实现，参与者更希望能有稍微不那么抽象的、更具体一些的实现。API对于这个任务的表示与参与者的预期大相径庭。也就是说，参与者不得不调整自己对于文本文件的理解，才能认识到`StreamWriter`和`StreamReader`正是用来解决这类问题的类。

我们在绝大部分的参与者中都观察到了这个情况。即便他们正在浏览的就是正确的、可以让他们解决问题的类（如`StreamReader`或`StreamWriter`）的文档，也会由于和他们所想的不同而觉察不到。所以问题的关键不是在文档中缺乏示例代码，而是参与者常常在文档中搜寻根本不存在的信息。他们希望能找到文本文件的具体表达方式，但是却只找到文件流的表达方式。

认知维度框架给了我们一套语言，使得我们可以用这套语言来更好地分析理解研究中观察到的问题。通过这次分析，我们使开发团队认识到：解决易用性问题的最好办法，除了改善文档质量以外，还需要设计和创建一个可以具体地表示硬盘上的文件系统以及文件的新类。

半年后，开发团队设计并实现了这个新类的原型版本。使用这个新类（`FileObject`）来完成和此前同样的任务所需的代码如下：

```
Dim f As New FileObject
f.Open(OpenMode.Write, "testfile.txt")
f.WriteLine("A line of text")
f.Close()
```

在这个例子中，每个`FileObject`类的实例都代表着一个文件。`FileObject`类可以有多种模式（在这个例子中，`OpenMode.Write`使得打开的文件可写）。这和此前的例子，即用不同的类来表示不同的模式（如`StreamWriter`和`StreamReader`），形成了鲜明的对比。

然后我们召集了一群有着相同背景（但并不是同一批）的参与者来进行类似的研究，任务相同，只是这次使用的是`FileObject`的API。这次的结果就明显不同了。在20分钟内，无需浏览任何文档，所有的参与者都顺利完成了任务。

29.4　使用不同的工作风格

我们的研究不仅使得这个重要的API加入了一个新的类，也使得整个微软对于研究产生了新的尊重和热忱。在每个.NET框架的API发布之前必须经过的审核流程中，加入了API易用性研究的步骤。

虽然这种搜集易用性信息的方法不断上升的关注度使我们大受鼓舞，但是此时我们仍然不清楚，开发人员对于API的易用性是各说各话，还是有一种统一的量化标准。我们选择参与文件I/O研究的开发人员的背景都相同：有经验的VB开发人员，每周使用VB超过20个小时。但是不同背景的开发人员会否做出同样的反应呢？

我们的预感是——不会。我们研究的参与者只能代表在研究VS IDE的时候我们设定的三种开发人员角色中的一种[①]。

- 机遇型开发人员

  特点是麻利地试验各项功能、以完成任务为重点，以及广泛使用顶层、具象的组件。

- 务实型开发人员

  特点是以代码为重点，并使用各种工具（如重构工具、单元测试工具等）来帮助他们确定代码的可靠性和正确性。

- 系统型开发人员

  特点是步步为营的开发方式，以及在开始开发之前就想要对于技术有一个透彻的了解

在我们的文件I/O研究中的开发人员差不多属于机遇型开发者。他们在第二次研究中使用`FileObject`类的方式几近完美地诠释了何为机遇型编程。这和他们在第一次研究中必须使用的更为抽象的类形成了强烈的对比。事实上，我们对于文件I/O的研究的主要结论之一就是，如果没有这样一些顶层的、具象的组件，机遇型开发人员便很难顺利完成任务。

而随着对API易用性的关注度不断走高，我们开始需要确定什么样的API才能使务实型和系统型的开发人员也满意。

这三种角色是由我们在此前多次研究中对开发人员的观察得来的，有着大量的数据作为基础。我和一位同事一起回顾了这些数据，并针对不同的角色根据认知维度框架来定义相应的API易用性。

首先，我们为每个维度定义了一套评级法。例如，对于抽象等级这一维度，我们将其分成了从“原生”到“整合”的数个等级。

抽象等级为“原生”的API所展示的是一些底层的组件。如果想用原生的API来完成任务，开发人员将需要选择一些原生的组件并相互配合着来达到想要的效果。

例如，原生的文件I/O的API就可能把文件表示为一个流。要从流中读取数据，API可能会提供一个`StreamReader`类。开发人员需要组合使用这些类，才能从文件中读取数据。例如：

① 关于我们如何确立这三种角色的介绍，参见参考文献[2]。

```
File f = new File();
StreamReader sr = f.OpenFile("C:\test.txt");
Byte b = sr.Read();
string s = b.ToString();
f.CloseFile();
```

在这样一个API中，开发人员需要知道如何来组合这些原生组件来达到想要的效果（在这个例子中，从文本文件中读取数据）。

与此相反的是，一个“整合”的API可能包含了一个代表文本文件的类，并提供对于文本文件的所有可用的操作。例如：

```
TextFile tf = new TextFile("C:\text.txt");
string s = tf.Read();
tf.Close();
```

在定义了这些维度的评级方法之后，我们使用了从易用性研究中得到的证据，来确定不同开发人员角色对于不同易用性维度的偏好。

在确定了偏好之后，我们回顾了每次研究中的录像以及观察参与者对不同API的使用情况。由于在征募参与者的面试的时候我们已经将每个参与者都和角色对应起来了，所以我们可以估计每个易用性维度的角色偏好。虽然我们并没有每次都对在研究中观察到的行为做量化分析，但我们仍然确信在面试时对于开发人员角色的主观判断是相当准确的，因为我们此前对此做了很多研究。而且我们还录下了参与者在进行试验时的解说词，这些常常使得他们的角色趋向非常明显。

我们观察到了这样一些角色趋向，比如，最机遇型的开发人员似乎比较喜欢“整合”的抽象等级。而系统型开发人员更喜欢原生组件提供的灵活性。务实型开发人员通常青睐我们所说的“分解式组件”（factored components），这种组件实际上就是把整合式组件的“模式”分解到不同的组件中。

我们对每个维度都一一进行分析，直到我们为每个开发人员角色都找到了一套对应的偏好。然后我们将这些数据显示在了图29-1中。

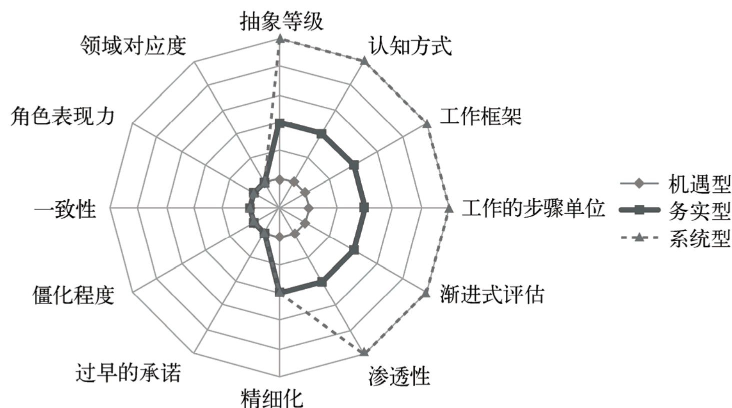


图29-1　每种角色对于认知维度的偏好

该图清楚地表明，每种角色对于API的易用性有着不同的观点。机遇型和系统型的开发人员所偏好的抽象等级的区别，和他们在学习方式、工作框架的区别相同。

在微软人们立刻就发现这个图的意义所在：要想设计和创造一套可以同时满足三种不同角色对易用性的诉求的API，即便有可能，也会非常困难。但这里就有个大问题了。大部分设计和编写API的团队都认为，他们应该针对所有三种角色来设计API，这一点完全正确。如果他们想要每种角色的开发人员都能舒服地使用他们的API，应该怎么做？他们应该为每种角色都做一套API吗？这就要说说我们研究的最后一步——基于场景的设计了。

基于场景的设计

当我们提到API这个词的时候，常常指的是在应用程序和程序员之间形成接口的全套的类。换句话说，每个开发人员有权使用的类，无论他是否真正使用，都被视为是API的一部分。当API设计团队说，他们想要创造一个三种角色的开发人员都愿意使用的API的时候，团队假设这些开发人员将会用到全部的API。

而我们从ATL Server的卡片分类法研究中已经确定API的易用性并不是由分析API中的所有类来决定的。API设计团队考虑的是全套的类，而开发人员只会想到他们使用这些API时的不同场景以及在这些场景中的使用体验。

我们在文件I/O的研究中实际看到了这一点。在那次研究中，我们发现参与者并没有花心思去了解.NET框架中的每一个类。相反，他们只关注那些能帮他们完成任务的类。此外，我们发现他们忽略了一些不符合他们预期抽象等级的类。“场景”和“预期”的结合帮助筛选出了他们认为有用的类。

所以，解决API在三种角色中的普遍易用性问题的重点，就在于使用API的场景以及可能会在这些场景中进行开发的开发人员类型。在很多情况下，场景可以决定开发人员的角色类型，而我们也就能确定在这种场景下应该使用哪种API的设计方案。

为了演示说明，我们可以考虑一下一个通用I/O的API。可能会用到这个API的场景包括：

- 从文本文件中读取一行文字；
- 写入二进制数据到文件；
- 执行内存映射的 I/O。

要说三种角色的开发人员都会涉及所有这些场景，这不太可能。比如，相比系统型开发人员，机遇型开发人员可能和文本文件的互动更多，因为文本文件通常可以使我们快速地建立原型和进行功能试验（文本文件可以由文本编辑器快速地创建和修改，所以要做测试案例、模拟数据等，都很简单）。

而务实型开发人员更容易和二进制数据打交道，因为他们所写的代码和硬盘上的数据有着更加直接的对应关系。由于是以代码为重点，务实型开发人员更喜欢那些可以帮他们减少数据表现形式的数量的框架。

最后，系统型的开发人员使用内存映射文件I/O的时候较多，因为在这样的抽象程度下可以更灵活。比如，使用内存映射的I/O，他们可以在多个进程之间共享内存，还能对内存的访问权

限有完全的控制。

在这个简单的例子里，三种角色确实都使用了I/O的API，但是更重要的是，他们使用了API的不同部件。我们可以根据使用者的不同来设计这些部件。比如，文本文件相关的组件可以是整合型的，而内存映射的组件可以是原生的。

在基于场景的设计中，API设计师们先写出他们希望开发人员在解决某个问题场景时应该写的代码。在这个过程中，他们需要（用认知维度的方法）考虑开发人员的偏好。也就是说，他们会把自己当作开发人员，站在开发人员的立场来想问题。在每个步骤中，API设计团队都可以使用认知维度框架来作为一个检查列表，来确认他们设计的路线是否正确。他们可以审阅他们所写的每个代码样例，以确认可以满足开发人员对于目标场景的偏好。只有当设计团队已经对这个问题场景的解决方案很满意了的时候，他们才开始设计和创建相应的API。

当然，一旦进入实现阶段，预料之外的设计问题就会出现。在这些情况下，设计团队可以回顾他们正在创建的API所对应的场景，并考虑这些设计问题对这些场景的影响。只要这样做，设计团队就可以真正关注多API的*用户体验*，而不是去关心整个API的*架构*。只要能努力满足各种场景五花八门的API需求，那在整个API的开发过程中，就可以基于这些对于API易用性的深入理解来做各种设计决定。

一个基于场景的设计方法是让API设计师去思考“API是做什么的”，而不是“API是什么”。这就清楚地表明，设计师所设计的，是开发人员使用这些类的体验，而不是API的整体架构。当考虑API的设计的时候，设计师可以专注于API的使用场景以及使用API的开发人员类型。

29.5 结论

在我们刚开始研究API易用性的时候，我们并不知道这个研究应该怎么做，也不知道这些努力是否值得。但我们现在可以毫无保留地说，API的易用性研究不但是完全可能，而且是至关重要的。关于不同类型的开发人员对API的使用方法的区别，以及他们对于API的预期的区别，我们已经学到了很多。我们不但可以使用这些认识来设计新的API，还可以解决已有API的易用性问题。

例如，基于场景的设计方法在微软已经使用了几年了。我们的API设计准则和审查流程更加强调了API的易用性研究以及基于场景设计[4]的必要性。

但这只是开始。还有很多事情需要做。由于在API正式发布之前找出API的易用性问题非常重要，我们还需要研究更多的方法，以帮助我们在API开发过程中尽早地发现易用性的问题。例如，Farooq等人描述了一种评估API易用性的方法，这种方法无需易用性研究[6]。

同样重要的是，要了解应用程序开发中的趋势对于API易用性的影响。例如，应用程序开发中有一种使用设计模式（如依赖注入）的趋势[7]。对于采用这类模式，在架构上我们已经有了非常明确和合理的理由，但是这些模式会对易用性有什么影响？Jeff Stylos调查了这些问题，并发现某些流行的设计模式有可能对易用性造成惊人的影响[5][10]。

有幸可以提出并回答这些问题让人感觉十分愉快。现在，我已经加入VS用户体验团队超过

10年了，我能体会到，关于如何提升开发人员的用户体验，我们已经学到了很多。我期待在今后能看到更多类似的发展。

29.6　参考文献

[1] [Blackwell 2002] Blackwell, A. F. 2002. First Steps in Programming: A Rationale for Attention Investment Models. *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (*HCC'02*): 2.

[2] [Clarke 2007] Clarke, S. 2007. What Is an End-User Software Engineer? Paper presented at the End-User Software Engineering Dagstuhl Seminar, February 18-23, in Dagstuhl, Germany.

[3] [Clarke and Becker 2003] Clarke, S., and C. Becker. 2003. Using the cognitive dimensions framework to measure the usability of a class library. *Proceedings of the First Joint Conference of EASE & PPIG* (*PPIG 15*): 359-366.

[4] [Cwalina and Abrams 2005] Cwalina, K., and B. Abrams. 2005. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Upper Saddle River, NJ: Addison-Wesley Professional.

[5] [Ellis et al. 2007] Ellis, B., J. Stylos, and B. Myers. 2007. The Factory Pattern in API Design: AUsability Evaluation. *Proceedings of the 29th International Conference on Software Engineering*: 302-312.

[6] [Farooq and Zirkler 2010] Farooq, U., and D. Zirkler. 2010. API peer reviews: A method for evaluating usability of application programming interfaces. *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*: 207-210.

[7] [Fowler 2004] Fowler, M. 2004. Module Assembly. *IEEE Software* 21(2): 65-67.

[8] [Green and Petre 1996] Green, T.R.G., and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of VisualLanguages and Computing* 7(2): 131-174.

[9] [Robillard 2009] Robillard, M. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26(6): 27-34.

[10] [Stylos and Clarke 2007] Stylos, J., and S. Clarke. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. *Proceedings of the 29th International Conference on Software Engineering*: 529-539.

第30章

“10倍”意味着什么？编程生产力的差距测量

Steve McConnell

在软件工程研究中，被验证得最多的结论就是对于同等经验的两个不同程序员，在效率和质量上可能会有10倍的差距。研究人员还发现，这种差距也适用于团队级别上，也就是说在同一行业内的不同的团队也是如此。

30.1 软件开发中的个人效率的变化

首先发现不同人在编程生产力上的巨大差距的研究，是1960年由Sackman、Erikson以及Grant三个人完成的[12]。他们研究了工作经验平均在7年的专业程序员，并发现最好和最差的程序员写新代码的时间比为20：1；调试次数是25：1；程序大小是5：1；程序的执行效率是10：1。他们还发现，程序员的经验和代码质量或效率并没有关系。

在详细地研究了Sackman、Erickson以及Grant的研究结果之后，我们可以发现他们所使用的方法中有很多缺陷，例如把使用低级程序语言和高级程序语言的程序员合在一起研究等。但是，即便把这些缺陷考虑进来，他们的数据也仍然表明，最好的程序员和最差的程序员之间的差距能达到10倍以上。

那次研究之后，还有很多其他关于专业程序员的相关研究都证明了一个结论：程序员的水平也分三六九等（具体内容参见参考文献[6]、[11]、[9]、[7]、[5]、[2]、[14]和[4]）。

除些之外，很多轶事传闻也支持这种观点。在20世纪80年代中期，当我还在波音公司工作的时候，有个约80个程序员组成的项目组正面临着无法按时完成一项关键任务的风险。这个项目对于波音来说至关重要，所以他们把项目上80个人中的一大半换成了另外1个人，而这位仁兄单枪匹马地完成了所有的编程工作，并按时交付了软件。我并没有在这个项目组中工作，也不认识这位天才，但是这个故事是一位我所信任的人告诉我的，所以我相信这是真的。

这种差距并不仅限于软件行业。Norm Augustine的一份研究指出，在各行各业中，包括写作、橄榄球、发明、警务工作等，都存在一个情况，那就是行业中位列前20%的顶尖人才的产出占到

了该行业总产出的50%，无论这些产出是得分、专利、侦破的案件还是软件[1]。你可以想想看，这还是有道理的。我们都知道，有的学生就是比其他学生优秀，运动员、艺术家甚至家长也是如此。既然这种差别存在于所有人群中，那么软件开发又怎么会例外呢？

30.1.1 巨大的差距带来的负面影响

Augustine的研究发现，由于有些人完全没有任何实质的贡献（例如不能得分的前锋、没有专利的发明家、无法破案的警探等），人与人之间的差距的实际情况可能比上文提到的数据还要大。

在软件行业中似乎就是这样。在多个已发表的关于软件开发效率的研究项目中，大约有10%的实验参与者无法完成实验任务。这些研究报告常常会这样说道：“所以我们从数据集中排除了这些参与者的结果。”但是在现实生活中，如果某个人“无法完成任务”，你就不能简单地“从数据集中排除他们的结果”了。你或者得等他们完成，或者得另外指派一个人完成他们的工作，等等。这里有一个有趣（而又可怕）的暗示，那就是在软件行业中，差不多有10%的人对项目产出的贡献是负数。

和此前一样，这也和我们在现实生活中的感受一致。我相信很多人都能够在共事过的人中找出符合这个描述的人。

30.1.2 什么造就了真正的“10 倍程序员”

很多人并不喜欢被贴上“10倍”这样的标签，因为他们觉得人们会说：“我们团队中曾有个超级程序员，他牛哄哄的，每个人都不愿和他来往，要是没有他整体效率反而还要高些。”

通常来说，任何对10倍程序员的实用定义都必须考虑这样的程序员对于团队其他人员的影响。我也知道的确有牛哄哄的超级程序员。但更多的时候，那些牛哄哄的超级程序员其实只是普通水平的一般程序员而已，甚至还达不到普通水平。他们只是用牛哄哄的外表来让自己的表现看上去不那么差而已。我所共事过的真正的超级程序员们除了技术水平以外，通常还有很好的团队精神（虽然有时也有些例外）。

30.2 测量程序员的个人生产力的问题

由于很多研究都指出不同程序员的效率可以有10倍的差距，导致很多人产生了一个想法，那就是测量他们在自己组织内的个人效率。无论如何，这种想法所涉及的测量“活的”程序员的生产力和一般研究中所说的生产力有很大不同。

软件工程研究通常用完成某个任务所需的时间、每小时或每个月能写多少行代码或者其他一些标准来测量生产力。但如果你尝试在商业环境中用这些标准来测量生产力，那就会碰到很多问题。

30.2.1 生产力=每月产出的代码行数吗

软件设计是一件非确定性的事情，对同样的一个问题，不同的设计师/开发人员会做出完全

不同的解决方案。如果我们用每月产出的代码行数（或者类似的标准）来衡量生产力，那么我们就默认了用10倍的代码来解决同样的问题的程序员就有10倍的生产力。显然事情并非总是如此。比如某个程序员可能会有一个绝妙的设计想法，结果只用10%的代码就解决了普通程序员需要100%的代码才能解决的问题。

有人曾断言，伟大的程序员写的代码总是更简短。事实上，编程水平和代码的简洁性之间可能有着某种关联，但我现在并不想做这样一个宽泛的结论。我只想说，伟大的程序员总是努力把代码写得更清楚，而结果通常就是更简短的代码。不过有时候，最清楚、最简单和最明显的设计和那些更"巧妙"的设计相比，需要更多一点的代码。在这种情况下，我认为伟大的程序员也会用稍微多一点的代码来避免太过于取巧的设计。无论怎么说，用"每月产出代码行数"来衡量生产力的想法都是有问题的。

Dilbert漫画中有一个故事：老板说每发现一个bug就奖励10块钱，大家都高呼这次赚到了，还有人想通过这个办法"写出辆小货车"来[①]。故事正好说明了这个问题，即如果你用代码的产出量来衡量生产力，有的人就会利用这一点，写很多很多也许完全没有必要的代码。这里的问题并不在于"代码量"这个标准，而在于旧式的管理思想，即"人们只会做会被考察的事情"。但你必须小心不要考察错东西。

30.2.2 生产力=功能点吗

"每个月的代码产出"所带来的问题有一部分可以依靠功能点的标准来衡量程序规模。功能点是一套"合成"的测量程序大小的标准。包括输入、输出、查询、文件数量等都被考虑进来，作为确定程序大小的参数。低效的设计/编程风格并不能产生更多的功能点，所以功能点这个标准不涉及代码量的问题。但是它却有一个更实际的问题，那就是你需要专业人士来计算功能点（很多公司并无这种人才），而且功能点和个人产出的对应也非常粗略，所以无法用于确定程序员的个人生产力。

30.2.3 复杂度呢

管理者常说："我总是把最困难、最复杂的编程任务交给最好的程序员去做。所以无论用什么方法来衡量，他的生产力好像总是比别人低，但是如果同样的事情让别人来做，就可能花上两倍的时间。"这种现象很正常，但是也会影响我们定义和测量生产力的方式。

30.2.4 到底有没有办法可以测量个人生产力

前文提到的这些困难让很多人认为：要想测量个人生产力简直困难重重，没人可以做到。但我认为要想正确地测量个人生产力是可能的，只是需要注意以下几点。

- 不要指望仅用一个单独的衡量标准就能了解个人生产力的实际整体状况

① 见http://dilbert.com/strips/comic/1995-11-13/。——译者注

你可以参照一下那些在体育比赛中搜集的统计数据。我们甚至无法用一个单独的标准来确定棒球比赛中击球手的水平。我们必须考虑打击率、全垒打、跑垒得分、上垒率以及其他种种因素。而且仅有这些数据还不够，我们还得去证明这些数据的意义。如果击球手的优劣无法用简单的标准来评断的话，难道程序员的个人生产力这样复杂的事情就可以吗？我们应该用的不是一个单独的标准，而是一整套标准的组合。这套组合标准的任务，就是让我们对个人生产力有更深入的了解。比如，这套标准可能包括准时完成任务的百分比、管理者的评分1～10、同事的评分1～10、每个月产出的代码行数、每行代码的平均缺陷数量、不当修复的比率，等等。

- 不要认为只要有了某种标准（无论单独或者组合）就可以对不同个体的生产力进行细致的鉴别了

  要记住一点：这些个人生产力的标准只是为你找出问题，但是并不会回答这些问题。对这些标准的不当使用，比如用来进行绩效评估，不但会带来管理上的问题，也会造成统计上的问题。

- 整体的趋势常常比某个时间点上的测量结果更重要

  将这些测量结果在不同个体间进行横向比较往往是得不出任何有意义的结论的。更有用的做法应该是将某个人的测量结果进行纵向分析，看看这个人有没有随着时间的推移而进步。

- 你要问自己：我测量个人生产力到底是为什么

  在研究环境中，研究员们需要评估不同技术的效率，所以需要测量个人生产力。相对于研究环境，在现实项目中使用同样的测量标准产生的问题就要多得多了。在现实项目环境中，你想要用这些测量标准来做什么？绩效评估？这主意不行，原因刚刚才说过。分配任务？但我所访问过的大部分管理者都说他们不必测量也知道谁是他们团队中的明星成员，这一点我也相信。做预算？不行，不同设计方法导致的差距、不同的任务难度以及其他相关的原因使得我们无法有效利用这些标准来做项目预算。

在现实项目中，个人生产力的标准很难找到一个对项目管理有益而又符合统计学规则的用处。根据我的经验，除了做研究之外，人们想对个人效率进行测量的动机通常来自一些在统计学上不能成立的结论。也就是说，虽然我知道在研究中对个人效率的测量非常有意义，但是我认为在实际项目中却很难找到它的合理用处。

30.3 软件开发中的团队生产力差距

软件专家们很早就已经发现，团队生产力的差距和个人生产力的差距一样大，是以数量级为单位的[11]。这里有一部分原因是因为物以类聚，人以群分，这一点已经由一次对来自18个组织机构的166个职业程序员的研究证明了[9]。

又比如，在一次对7个完全相同的项目的研究中，研究人员发现，在这些团队中耗费精力最多的是最低的3.4倍，而对于程序的大小，最大的是最小的3倍 [3]。虽然生产力有一定差距，但是这次研究中的程序员都来自相似的背景。他们都是科班出身的职业程序员，而且都有多年的经验。我们

可以合理地推测，如果研究对象的背景差异再大一些，那么他们之间的差距会更大。早期的一份对编程团队的研究曾指出，对于同样的项目，不同团队所提交的程序大小的比例可以达到5：1，而所需时间的比例可以达到2.6：1[15]。

Barry Boehm等研究人员为了确立COCOMO II①成本估算模型而研究了超过20年的数据，并总结到：对于同样的程序，能力评分在15+的团队需要的时间是得分为90+的团队的3.5倍（以100分算）。如果一个团队比另一个团队在程序语言或者应用领域上更有经验，那么这个差距还会更大。

一个具体的例子就是Lotus 1～2～3第三版和微软Excel 3.0开发团队之间的生产率差距。两者都是在1989～1990这个时间段发布的桌面电子表格应用程序。由于很少看到两个公司公布相似项目的数据，所以这种死对头之间的对比就显得尤其有趣了。这两个项目的数据如下：Excel的工作人员总共消耗了50个工年②，共写了649 000行代码 [8]。而Lotus 1～2～3消耗了260个工年，共写了400 000行代码 [13]。Excel的团队每个工年的代码产出是13 000行代码。而Lotus的团队每个工年的产出只有1500行代码。两个团队之间的生产力差距超过了8倍，正好证明了我们此前的主张，即团队的生产力也有差距，并且有着更大量级的差距。

有趣的是，这些量化的结果和局外人对于这些项目的感觉非常贴近。Lotus 123第三版当时是出了名的跳票王，比预期的时间至少晚了两年才发布。而在微软内部，Excel大受赞扬，被誉为是微软最成功的项目之一。对于真实公司的真实项目，这种程度的同类比较恐怕已经是能做到的极致了。

如前所述，这个例子说明了造成生产力差距的各种因素。Lotus和微软都煞费苦心地为各自的项目招募了顶级的人才，所以我怀疑团队生产力的差距并不只是由于团队成员的能力差距造成的，还牵扯到了很多组织结构上的因素，比如产品远景是否清楚、客户需求是否明确以及成员之间是否能同心协力，等等。

组织的因素会影响团队生产力的发挥。杰出的组织中个人能力平庸的团队可以超越平庸组织中个人能力杰出的团队。当然，像杰出的组织+杰出的团队或者平庸的组织+平庸的团队这样的组合也不是没有的。在这种时候，团队生产力（或者叫组织生产力）和个人生产力一样相差10倍也就不足为奇了。

30.4 参考文献

[1] [Augustine 1979] Augustine, N.R. 1979. Augustine’s Laws and Major System Development Programs. *Defense Systems Management Review*: 50-76.

[2] [Boehm and Papaccio 1988] Boehm, Barry W., and Philip N. Papaccio. 1988. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering* 14(10): 1462-1477.

[3] [Boehm et al. 1984] Boehm, Barry W., T.E. Gray, and T. Seewaldt. 1984. Prototyping Versus Specifying: A Multiproject Experiment. *IEEE Transactions on Software Engineering* 10(3): 290-303.

① Constructive Cost Model II，一种软件成本估算的方法，基于以往项目的数据和当前项目的特征对开发成本进行预估。——译者注

② staff year，即每人每年可以完成的工作量。——译者注

[4] [Boehm et al. 2000] Boehm, Barry, et al. 2000. *Software Cost Estimation with Cocomo II*. Boston: Addison-Wesley.

[5] [Card 1987] Card, David N. 1987. A Software Technology Evaluation Program. *Information and Software Technology* 29(6): 291-300.

[6] [Curtis 1981] Curtis, Bill. 1981. Substantiating Programmer Variability. *Proceedings of the IEEE* 69(7): 846.

[7] [Curtis et al. 1986] Curtis, Bill, et al. 1986. Software Psychology: The Need for an Interdisciplinary Program. *Proceedings of the IEEE* 74(8): 1092-1106.

[8] [Cusumano and Selby 1995] Cusumano, Michael, and Richard W. Selby. 1995. *Microsoft Secrets*. New York: The Free Press.

[9] [DeMarco and Lister 1985] DeMarco, Tom, and Timothy Lister. 1985. Programmer Performance and the Effects of the Workplace. *Proceedings of the 8th International Conference on Software Engineering*: 268-272.

[10] [DeMarco and Lister 1999] DeMarco, Tom, and Timothy Lister. 1999. *Peopleware: Productive Projects and Teams*, Second Edition. New York: Dorset House.

[11] [Mills 1983] Mills, Harlan D. 1983. *Software Productivity*. Boston: Little, Brown.

[12] [Sackman et al. 1968] Sackman, H., W.J. Erikson, and E.E. Grant. 1968. Exploratory Experimental Studies Comparing Online and Offline Programming Performance. *Communications of the ACM* 11(1): 3-11.

[13] [Schlender 1989] Schlender, Brenton. 1989. How to Break the Software Logjam. *Fortune*, September 25.

[14] [Valett and McGarry 1989] Valett, J., and F.E. McGarry. 1989. A Summary of Software Measurement Experiences in the Software Engineering Laboratory. *Journal of Systems and Software* 9(2): 137-148.

[15] [Weinberg and Schulman 1974] Weinberg, Gerald M., and Edward L. Schulman. 1974. Goals and Performance in Computer Programming. *Human Factors* 16(1): 70-77.

撰 稿 人

Jorge Aranda是维多利亚大学的博士后研究员，在SEGAL和CHISEL实验室工作。他之前在多伦多大学拿到了博士学位，学位研究包含对几十家软件组织的数百名专业人士进行的实证研究、观察和访谈。这些研究，以及他之前作为软件咨询师和开发者的经验，使他相信团队协作和团队交流是我们领域中最严重的问题，而他的研究也旨在解决这些问题。Jorge是加拿大不列颠哥伦比亚省维多利亚市的墨西哥人，与他的妻子和他们的猫生活在一起。他喜欢桌游、阅读、看电影、烹饪、跑步和变戏法。

Thomas Ball是微软研究院的首席研究员，他管理着软件可靠性研究小组（http://research.microsoft.com/srr/）。Tom在1993年拿到了威斯康星大学麦迪逊分校的博士学位。1993年~1999年，他在贝尔实验室工作。自1999年起，他一直在微软研究院。他是SLAM项目的发起者之一，SLAM是一个针对C语言的软件模型检查引擎，构成了静态驱动程序验证工具的基础。Tom兴趣广泛，从程序分析、模型检查、测试，到用自动化定理证明定义及度量软件质量的问题。

Victor R. Basili博士是马里兰大学计算机科学院的教授。他是弗劳恩霍夫（Fraunhofer）实验性软件工程中心的创始主任，现在他担任资深研究员，也是NASA/GSFC软件工程实验室（SEL）的创始人和主任之一。在过去超过35年的时间中，他用实证研究中观察和进化知识的机制度量、评估并改善软件开发的流程和产品。比如，“目标－问题－指标”法、质量改进范例、以及经验工厂。

Basili博士在德克萨斯大学拿到了计算机科学博士学位，也得到了两个荣誉学位。他曾经得到过ACM SIGSOFT、IEEE计算机学会、NASA等机构的奖励。在2005年，第27界国际软件工程大会为他举办了一次研讨会。2007年，他获得了弗劳恩霍夫（Fraunhofer）奖章。他是*IEEE Transactions on Software Engineering*（《IEEE软件工程会刊》）的主编，也是*Journal of Empirical Software Engineering*（《实证软件工程杂志》）的共同创始主编。

Andrew Begel是微软研究院“编程中的人际交互”小组的研究员，位于美国华盛顿雷德蒙。他研究微软的软件工程师，试图理解他们如何交流、合作、协调，以及这些活动对同地和异地开发效果的影响。研究之后，他会构建工具来减轻他们所发现的协调问题。Andrew引导过多个工作坊，包括软件工程中人文因素、计算机教育中的动觉学习、对计算机教学进行培训以及复杂系统科学。他在许多计算机科学和计算机科学教育相关的大会和工作坊中担任程序委员会成员。

Christian Bird是微软研究院实证软件工程小组的博士后研究员。他对大型开发项目中软件设计和社会动态的关系，以及它们对生产力和软件质量的影响最感兴趣。为了实证性地回答那个领域的问题，Bird博士开创了一系列软件挖掘技巧。他研究过微软、IBM和开源领域的软件开发团队，检验分布式团队、所有权政策、以及团队完成软件任务的方式所产生的影响。

Bird博士是ACM SIGSOFT优秀论文奖的获得者，以及加州大学戴维斯分校"最佳研究生研究员"称号获得者。而他的博士学位也是在加州大学戴维斯分校获得的，导师是Prem Devanbu。他在顶尖软件工程学术会中引起过人们的注意，在CACM中有著名的研究成果，也是BYU的国家优秀学者。BYU也是他拿到计算机科学学士学位的地方。

Barry Boehm博士是南加利福尼亚大学计算机科学和工程及系统工程部的TRW教授。他也是DoD-Stevens-USC系统工程研究中心的研究主任，及USC系统及软件工程中心的创始主任名誉教授。他在1989~1992年是DARPA-ISTO的主任，1973年～1989年是TRW的主任，1959年～1973年是Rand公司的主任，1955~1959年是General Dynamics的主任。他的贡献包括COCOMO家族的成本模型和螺旋家族的流程模型。他是各主要专业协会的资深委员，包括计算（ACM）、航空（AIAA）、电子（IEEE）、系统工程（INCOSE）。他是美国国家工程院院士，并由于在系统工程和系统科学领域的杰出贡献在2010年获得了IEEE的Simon Ramo奖章。

Marcelo Cataldo是卡内基梅隆大学软件研究学院的研究员。他的研究重点在理解大规模软件系统的结构和开发组织实现那些系统的能力之间的关系。调查的具体领域包括：（1）开发和分析适用于分布式软件开发的软件架构；（2）评估技术因素、社会组织因素、及其之间的相互作用对分布式项目开发生产力和软件质量的影响。Cataldo博士2007从卡内基梅隆大学计算机科学院获得了"计算、组织和社会"的硕士和博士学位。他也握有国家技术大学（阿根廷不宜诺斯艾里斯）信息系统学士学位，以及卡内基梅隆大学信息网络硕士学位。

Steven Clarke是微软开发部的资深用户体验研究员。他在1993年和1997年分别获得了苏格兰格拉斯哥大学的硕士和博士学位。1997年~1999年，他是摩托罗拉的软件开发人员，构建智能卡操作系统的开发工具。他从1999年开始工作于当前的职位。他与Visual Studio和微软研究院的同事们一起，使用针对开发习惯和工作风格的研究结果，来识别能大幅度提升开发者用微软的工具和平台构建应用程序的用户体验的方式。

Jason Cohen创办了四家公司，包括Smart Bear软件公司，该公司制造了最流行的同行代码审查工具Code Collaborator。他也是*Best Kept Secrets of Peer Code Review*的作者，书中包含了在录的最大的代码审查案例。现在，Jason在http://blog.ASmartBear.com为极客们写创业相关的文章，也正在运营他最新创办的公司WPEngine.com。

Robert Deline是微软研究院的首席研究员（http://research.microsoft.com/~rdeline），工作于软

件工程和人机交互的交叉领域。他的研究团队用以用户为中心的方式设计开发工具：他们研究开发团队来理解他们的工作实践并打造新的工具原型来改善那些实践。他在1999年从卡内基梅隆大学获得了博士学位，1993年从弗吉尼亚大学获得了学士和硕士学位。

Madeline M. Diep是弗劳恩霍夫（Fraunhofer）中心实验软件工程的研究科学家。她的研究重点包括软件质量保障、软件分析和实证评估。她从内布拉斯加大学林肯分校获得了计算机科学博士学位。

Hakan Erdogmus是加拿大渥太华的独立咨询师，卡尔加里大学计算机科学院的辅助教员，IEEE软件的主编。他的专长是软件流程、软件开发实践以及软件开发的经济学。1995~2009年，他是加拿大国家研究理事会的信息技术研究所的研究科学家。Hakan拥有蒙特利尔魁北克大学INRS电信专业的博士学位（1994），蒙特利尔麦吉尔大学计算机科学院的硕士学位（1989），伊斯坦布尔博阿齐奇大学计算机工程部的学士学位（1986）。他是IEEE和IEEE计算机学会的资深会员，也是ACM和敏捷联盟的会员。

Michael W. Godfrey是加拿大滑铁卢大学计算机科学David R. Cheriton学院的副教授，他也是软件架构小组SWAG的成员。在获得多伦多大学计算机科学博士学位后，他在康奈尔大学做了两年教员，然后在1998年加入了滑铁卢大学。在2001年到2006年间，他在北电和加拿大国家科学与工程研究理事会（NSERC）资助的电信软件工程院任工业研究副主席。他的研究重点涉及软件的进化：理解软件如何随着时间变化以及为什么。他对循证软件工程、软件克隆分析、挖掘软件知识库、软件工具设计、逆向工程和程序理解特别感兴趣。

Mark Guzdial是乔治亚理工学院交互计算学院的教授。他是计算机教学的主要研究者，帮助把计算机科学教育的需求从仅仅针对计算机专业转向了应对整个校园。最近他写了一系列名为《媒体计算》的教科书，通过处理数字媒体来介绍计算。

Jo E. Hannay拥有爱丁堡大学计算机科学基础的实验室逻辑、形式规范和类型理论的博士学位。他目前是挪威Simula研究实验室的研究员。他曾经在保险行业做过程序员，也在奥斯陆大学做过副教授。他的研究内容包括定义软件工程的专业技术和任务、开发并合并科学和实践理论、以及理解大型敏捷开发项目中发生了些什么。他曾经是一名小提琴演奏家。

Ahmed E. Hassan是加拿大安大略省金斯顿皇后大学超大规模系统软件工程部的NSERC/RIM工业研究主席。Hassan博士牵头组织并创立了挖掘软件仓库（MSR）大会以及相关的研究社区。MSR社区旨在用软件项目及其演进过程中的实证数据来支持决策流程。Hassan博士共同主编了《IEEE软件工程会刊》和《实证软件工程杂志》中关于MSR话题的特刊。

Hassan博士及其团队所开发的所有工具和技术都已集成于全球上百万人使用的产品中。Hassan博士的产业经营包括：帮助RIM的黑莓无线平台搭建架构，工作于IBM研究院阿尔马登实验室，工作于北电网络的计算机研究实验室。Hassan博士拥有全球好几个地区的专利，包括美国、欧洲、印度、加拿大和日本。

Isreal Herraiz是西班牙马德里智者阿方索十世大学（Universidad Alfonso X el Sabio）任教并进行研究。他在GSyC/Libresoft研究小组工作的同时，从胡安卡洛斯国王大学（Universidad Rey Juan Carlos）获得了博士学位。他的研究重心在软件演进、挖掘软件仓库、实证软件工程的交集上，侧重于软件项目的大规模研究和统计分析。他同时也是好几个免费开源软件项目的活跃贡献者。

Kim Sebastian Herzig是德国萨尔大学（Saarland University）软件工程实验室Andreas Zeller教授手下的研究生。他当前的研究活动重点是实证软件工程和挖掘软件仓库。他正在探索和分析版本归档和缺陷数据库来创建能帮助软件开发者开发可靠的软件并能够估计源代码改变风险的工具和技巧。

Cory Kapser已经发布了关于编码实践的众多文章，特别是关于代码克隆的文章。他赢得了2006年第13届IEEE逆向工程工作会议的最佳文章奖。

Barbara Kitchenham是英国基尔大学定量软件工程的教授。她在软件工程的产业和学术领域已经工作了超过30年。她的主要研究重点是软件度量及其在项目管理、质量控制、风险管理和软件技术评估中的应用。她最近的研究侧重于把基于证据的实践应用于软件工程。她是一位特许数学家、数学及应用数学研究所的研究员、皇家统计学会研究员、以及IEEE计算机学会成员。

Andrew Ko是华盛顿大学信息学院的副教授。他的研究方向包括软件开发设计中人与合作的方面，更广泛地说，是人机交互和软件工程领域。他所发布的文章涵盖了所有这些领域。他曾获得过顶级会议的最佳文章奖，比如软件工程国际大会（ICSE）及ACM计算中的人性因素大会（CHI）等。他所开发的调试工具Whyline也得到了众多的点击，它允许用户在遇到问题输出时，能够问“为什么”。2004年，他同时获得了NSF和NDSEG研究奖学金以支持他的博士研究。他在Brad Myers的指导下从卡内基梅隆大学人机交互学院获得了博士学位。2002年，他从俄勒冈州立大学计算机科学和心理学院获得了荣誉理学学位。

Lucas Layman是弗劳恩霍夫（Fraunhofer）实验性软件工程中心的研究科学家。他的专长领域包括把度量和指标运用于评估和提升软件开发流程和产品、敏捷方法和软件可靠性。他在好几个组织中进行过实证研究和技术转移，包括NASA、微软、国防部、IBM和Sabre航空解决方案。除了实证评估流程和流程改进之外，Layman博士也调研了软件开发中的人性因素并在计算机科学教学领域发表了几篇论文。他在2009年从北卡罗来纳州立大学获得了计算机科学博士学位，

2002年从Loyola学院获得了计算机科学学士学位。他也曾在微软研究院和加拿大国家研究理事会工作过。

Steve McConnell是Construx软件公司的CEO和首席软件工程师，他对各行业提供咨询、在研讨会上授课、并监管Construx软件公司的工程实践。他是*Software Estimation: Demystifying the Black Art*（2006）、*Rapid Development*（1996）、*Software Project Survival Guide*（1998）和*Professional Software Development*（2004）的作者，也发表了众多科技文章。他的书在软件开发杂志、游戏开发杂志、亚马逊主编和其他地方获得了多次“年度最佳书籍奖”。

Steve是IEEE软件杂志的名誉主编，SWEBOK的专家小组成员，也是IEEE计算机学会专业实务委员会的上任主席。1998年，《软件开发》杂志把他和Bill Gates 和Linus Torvalds一起推选为3个软件行业最具影响力的人物。

Tim Menzies是西弗吉尼亚大学计算机科学院的副教授，研究软件工程的数据挖掘算法。他的算法可以找到软件工程数据中的模式来预测软件质量方面的问题（缺陷、构建时间等）。他的工作被经常性地总结为“少即是多”，AI推动工具可以快速自动地学习最少的约束条件最大程度影响关键决策。

NASA的上任研究主席Menzies博士拥有澳大利亚新南威尔士大学的人工智能的博士学位（1995）。他是超过190篇被经常引用的文章的作者，PROMISE软件工程重复实验系列大会的共同创始人（http://promisedata.org/data ）。最近，他在超过26000位软件工程研究者中被排名为最优秀的1%（http://academic.research.microsoft.com/CSDirectory/author_category_4_last5.htm）。他的网站是：http://menzies.us。

Gail Murphy是英属哥伦比亚大学（UBC）计算机科学院的教授。她在1996年获得了华盛顿大学的硕士和博士学位之后加入了UBC。在读研究生之前，她在一家电信公司作了5年软件开发人员。她也握有阿尔伯塔大学的学士学位。

Murphy博士主要工作于构造更简单有效的工具来帮助开发人员管理软件演化的任务。2005年，她拿到了UBC的Killam研究奖学金，并获得了AITO Dahl-Nygaard青年奖以表彰她对软件演化的贡献。2006年，她获得了NSERC的Steacie奖学金和CRA-W Anita Borg早期职业奖。2007年，她帮助共同创办了Tasktop科技有限公司，并现任董事会主席和CFO。2008年，她出任ACM SIGSOFT FSE大会的程序委员会主席并获得了华盛顿大学工程学院的早期职业钻石奖。2012年，她会担任软件工程国际大会的程序联合主席。她目前是《ACM软件工程会刊》杂志的副主编，也是《ACM通讯》的编辑委员会成员。她事业中最有收获的事情是有机会与许多有才华的本科生与研究生协作。

Nachiappan Nagappan是微软研究院软件可靠性研究小组的资深研究员。他的主要研究领

域是实证软件工程。他已经与微软的若干团队合作过，把他的想法付诸实践。他的博士学位是在北卡罗来纳州立大学获得的。

Andy Oram是O'Reilly媒体的编辑。他在O'Reilly的工作包括2005年开创性书籍*Running Linux*、影响2001年的标题*Peer-to-Peer*、2007最畅销的《代码之美》（*Beautiful Code*）。Andy也经常为O'Reilly网络和其他出版物写东西，关于互联网相关的政策问题和影响技术创新及其社会影响的趋势。体现他工作的印刷出版物包括《经济学家》、《ACM通讯》、*Copyright World*以及《互联网法律与商业》。

Thomas Ostrand是AT&T新泽西弗洛厄姆公园实验室信息和软件系统研究部的首席技术人员。他当前的研究兴趣包括软件测试的生成和评估、缺陷分析和预测、软件开发和测试工具。

加入AT&T之前，Ostrand博士和西门子公司研究部、Sperry Univac、罗格斯大学计算机科学院合作过。他是ACM的高级会员，也是ACM/SIGSOFT执行委员会的高层委员。他曾做过软件测试和分析国际研讨会以及软件工程预测模型PROMISE大会的程序主席和督导委员会成员。他目前是《实证软件工程》杂志的副主编，也是《IEEE软件工程会刊》的前任副主编。

Dewayne E. Perry是德克萨斯大学奥斯汀分校软件工程部的摩托罗拉校董主席。他的前三分之一软件工程生涯是一名专业软件开发者，然后合并了研究（卡内基梅隆大学计算机科学院的客座教员）、软件架构以及设计咨询。之后的16年他在新泽西默里山的贝尔实验室做软件工程的研究。他从2000年1月开始在德克萨斯大学奥斯汀分校任职。

他在实证研究、软件流程的正规模型、流程和产品支持环境、软件架构及形式和技术规范方面进行了开创性的研究。他对架构在多站点软件开发中的地位及其在产品线中投资公司软件资产的角色特别感兴趣。

他是ACM SIGSOFT和IEEE Computer Society（IEEE计算机学会）的会员。曾担任过各种软件工程大会的组织主席、程序主席和程序委员会成员。他曾经是*Wiley's Software Process: Improvement & Practice*（《Wiley软件流程：改进和实践》）的共同主编，也是《IEEE软件工程会刊》的前任副主编。

Marian Petre是英国开放大学（Open University）的计算机教授和计算研究中心的主管。她拥有皇家学会沃尔夫森优秀研究奖，以表彰她在软件设计方面的实证研究。她的研究侧重于专业软件设计中的专家论证和代表性。她也是实证研究方法的创新教育者，编写了几本书籍，包括*The Unwritten Rules of PhD Research*（《博士研究生的不成文规定》，现已第二版）以及*A Gentle Guide to Research Methods*（《研究方法的温柔指南》）。

Lutz Prechelt是柏林自由大学软件工程院的教授。他的研究生涯从人工智能、神经网络学习、编译器构造和并行计算开始，之后转移至实证软件工程。他也曾做过软件开发人员和软件开发经理。现在，他的研究重点在软件流程的定性和定量研究上，特别是敏捷流程，以及研究方法论和研究质量。

Rahul Premraj是阿姆斯特丹自由大学计算机科学院的副教授。他2002年从英国罗伯特戈登大学获得了信息系统硕士学位，2007年从英国伯恩茅斯大学获得了博士学位。他的研究兴趣包括实证软件工程、软件档案挖掘、软件质量保障、分布式软件开发和软件流程改进。

Forrest Shull博士是马里兰弗劳恩霍夫（Fraunhofer）实验性软件工程中心（FC-MD）的分区主任。该中心是一个非盈利的研究和技术转移组织，他在那里带领度量和知识管理分区。他也是马里兰大学帕克分校的兼职副教授。

在FC-MD，他曾是NASA办公室安全和任务保障、NASA安全中心、美国国防部、美国国家科学基金、国防部高级研究计划局以及摩托罗拉和富士通美国实验室等项目的首席研究员。他也为NASA的工程师开发并教授了几个软件度量和检查的课程。

从2007年起，Forrest担任*IEEE Software*（IEEE软件）实证结果部门的副主编和热门的“Voice of Evidence”（证据之声）部门的编辑。他将在2011年就任《IEEE软件》的主编。他也是《实证软件工程杂志》的编辑部成员。

Beth Simon是加州大学圣地亚哥分校计算机科学与工程系的教员。她的研究着重在计算机科学的教育上，特别对（大学和产业界的）计算机新手、同行指令以及评估感兴趣。Simon博士在代顿大学获得了计算机科学的学士学位，在加州大学圣地亚哥分校计算机科学与工程系获得了硕士和博士学位。

Diomidis Spinellis是希腊雅典经济与商业大学管理科学与技术系的教授。他的研究兴趣包括软件工程、计算机安全、编程语言等。他写了两本获奖的*Open Source Perspective*（开源视点）书籍：*Code Reading*（《代码阅读》）和*Code Quality*（《代码质量》），也写过几十篇科技文章。

Spinellis博士是《IEEE软件》的编辑委员会成员，专栏“Tools of the Trade”（贸易工具）的作者。他也是FreeBSD的提交者，UMLGraph及其他开源软件包、库、工具的开发者。他拥有软件工程的工程硕士和计算机科学的博士学位，均在伦敦帝国学院获得。Spinellis博士是ACM和IEEE的资深会员，也是Usenix协会的会员。

Neil Thomas是Google的软件工程师。他工作在HTML5的移动应用程序上，包括Gmail和Google Buzz，试图持续推进移动网络并重新定义网络浏览器的可能性。他曾获多项专利。他也置身于Google的内部工具研发，来改善程序员的生产力。

Thomas先生2010年从英属哥伦比亚大学（UBC）得到了计算机科学的学士学位。在他大学

的最后一年，他有幸与Gail Murphy一起工作于软件实践实验室。他的研究探索了开发人员如何理解并推敲复杂系统的模块化。

Walter F. Tichy从1986年开始在德国卡尔斯鲁厄理工学院（前卡尔斯鲁厄大学）任计算机科学教授，2002年至2004年担任计算机科学院院长。他也是FZI（卡尔斯鲁厄的一个技术转移中心）的主任。他的主要研究兴趣是软件工程和并行化。20世纪80年代，他创立了修订控制系统（RCS），一个基于CVS的版本管理系统，至今仍在全球范围内使用。Tichy博士在学生时代初次接触了并行计算，那是由16个PDP-11计算机组成的并行计算机C.mmp在CMU问世的时候。他在20世纪80年代也工作于连线机和一系列并行计算机上，并和其他学生一起开发了Parastation——能运行2009年6月500强中排名前十的机器。

Tichy博士在1980年从卡内基梅隆大学获得了计算机科学博士学位。他的论文是第一篇讨论软件架构的。由于他坚持软件研究者需要用实证研究测试他们的断言而不是依赖直觉，因而引发了争议。他进行过众多对照实验，测试类型检查、继承深度、设计模式、测试方法和敏捷方法等对程序员生产力的影响。

Burak Turhan是芬兰北部奥卢大学信息处理科学系的博士后研究员。在移居芬兰之前，他是加拿大国家研究理事会信息技术研究所的副研究员。他拥有伊斯坦布尔博阿齐奇大学计算机工程博士学位。他的研究重点包括软件质量的实证研究，在软件工程中运用机器学习和数据挖掘的方法来进行缺陷和成本模型，敏捷、精益软件开发（特别关注测试驱动开发）。

Elaine Weyuker是AT&T软件工程研究员。在来到AT&T之前，她是纽约大学Courant数学科学研究所的计算机科学教授。她当前的研究着重于软件缺陷预测、软件测试、软件指标和度量。在早前的生活中，Elaine研究过计算理论，与Martin Davis和Ron Sigal一起写了*Computability, Complexity, and Languages*（《可算性、复杂性和语言》）。

Elaine曾在2010年获得了ACM总统奖，在2009年获得了ACM SIGSOFT回顾性影响论文奖，2008年获得了Anita Borg学院技术领导奖，2007年获得了ACM SIGSOFT杰出研究奖。她也是美国国家工程院院士、IEEE院士、ACM院士，并因为杰出的软件工程研究获得过IEEE的Harlan Mills奖、罗格斯大学50周年杰出校友奖、AT&T主席多元奖、也被YWCA命名为“成就女性”。他是ACM妇女理事会（ACM-W）的主席，也是多元计算联盟的执行委员会成员。

Michele Whitecraft是一个充满活力的老师、讲师和研究员。她用一个整体的、跨学科的方法来执教，并积极推进妇女参与科学。她获得过优秀中学科学教育总统奖、Tandy学者教师奖、卓越教育总督奖。根据她在能源部、国家科学基金、国家环境健康科学研究院、国家卫生研究院、国家航空航天协会所获得的独特研究咨询经验，Michele把科学课程与真实世界的实验联系起来，实验范围从普林斯顿的国际热核实验反应堆到加州大学伯克利分校的人工嬗变超铀元素。

带着超过25年的高中和大学化学教学经验，她写了几篇专题论文来加强全国范围的科学教学，并在各大全国性的会议中做了演讲。她在*BioScience*、*Journal of Nuclear Materials*、*Human Ecology*和*Encyclopedia of Ethics*中都发表过文章。

Michele在这些国家级组织和研究项目中的经验激发了她帮助所有学科推进女性参与度的渴望，她试图实现NSF在2020年之前达到科学参与者性别比例50-50的目标。Michele是美国化学学会、美国先进科学协会、妇女工程师协会、美国大学妇女协会、课程开发与监督协会和国家科学教育研究协会的成员。Michele拥有课程与教学的硕士学位，目前是康奈尔大学学习、教学与社会政策的博士生。

Laurie Williams博士是北卡罗来纳州立大学的副教授。她获得了理海大学工业工程的本科学位，杜克大学的MBA，以及犹他大学计算机科学博士学位。在回到学术界读博士之前，她在IBM工作了9年。Laurie是*Pair Programming Illuminated*（《结对编程解析》）的第一作者，*Extreme Programming Perspectives*（《极限编程透视》）的共同编辑。她是极限编程世界大会的创始人，现在该会议演变成了敏捷软件开发大会。她进行了多次极限编程和敏捷开发实践的实证研究。

Wendy M. Williams博士是康奈尔大学人类发展系的教授，她在那里学习了智力的开发、评估、培训以及社会影响的多种形式。她写了9本书、编了5卷书、写了几十篇文章，包括2007年的APA书籍*Why Aren't More Women in Science*?，获得了2007年独立出版商图书奖，以及2010年与Stephen Ceci合著的牛津书籍*The Mathematics of Sex: How Biology and Society Conspire to Limit Talented Women and Girls*。

Williams博士获得了两次美国心理协会的早期职业奖，三次门萨研究基金会的资深研究者奖，也是各种专业学会的会员。她的研究在《自然》、《新闻周刊》、《商业周刊》、《科学》、《科学美国人》、《纽约时报》、《华盛顿邮报》、《今日美国》、《费城问询报》、《高等教育纪事》、《儿童杂志》等媒体出版物中均有刊登。2009年，她发起了国家卫生研究院资助的康奈尔大学妇女与科学研究所，研究和促进女性科学家事业的研究推广中心。

Greg Wilson是“软件木匠”项目的带头人，这是一个针对科学家和工程师的基本软件开发技巧速成班。在之前的生活中，他做过程序员、作者和教授。Greg有爱丁堡大学计算机科学的博士学位。他是一位自豪的加拿大人。

Andreas Zeller是德国萨尔大学的软件工程教授。他的研究关注大型软件系统的分析，特别是它们的执行和开发历史。他是自动化调试（“为什么我的程序会失败？”）和软件档案挖掘（“大部分的缺陷发生在哪里？”）的先驱。

Thomas Zimmermann从帕绍大学获得计算机科学文凭，并从德国萨尔大学获得了博士学

位。他是微软研究院软件可靠性研究小组的研究员，也是卡尔加里大学计算机科学系的兼职助理教授。他的研究兴趣包括实证软件工程、软件知识库挖掘、软件可靠性、开发工具和社交网络。著名的研究包括系统挖掘版本档案和缺陷数据库，以实施实证研究来构建工具支持开发者和管理者。

Zimmermann博士共同组织了ICSM关于软件工程之谜的工作会议（2007年MythSE），软件缺陷工作坊（2008年和2009年DEFECTS）和软件工程的推荐系统（2008年和2010年RSSE）。他获得了两次ACM SIGSOFT杰出论文奖，以表彰他在07年ICSE和08年FSE大会所发表的工作。他曾在各种程序委员会任职，包括ICSE、MSR、PROMISE、ICSM和ACM推荐系统大会（RecSys）。他是2010年和2011年MSR程序委员会的联合主席。他的主页是http://thomas-zimmermann.com。

Software Development/General Programming

软件之道 软件开发争议问题剖析

“虽然我们自称是‘工程师’，然而编程过程并非机械地由数据驱动，而是更多地取决于编程人员的感受。以软件开发的大量经验性数据为基础，编程过程完全可以达到个性化与系统化的统一。”

——Jason Cohen，Smart Bear和WPEngine公司创始人

相信大家常常听说某些工具、技术和实践方法可以改进软件开发，但其中哪些说法是可被证实的，哪些仅仅是人们一厢情愿的想法？本书收录了Steve McConnell、Barry Boehm和Barbara Kitchenham等几十位软件工程领域顶尖研究人员的文章，深入讨论了软件开发社区中常见的一些观点，一些是确凿事实，一些则是荒诞说法。他们的深刻见解定会让你大开眼界。

- 某些编程人员的工作成效果真是他人十倍之多？
- 测试驱动的开发果真能帮助更快、更好地开发代码？
- 软件的bug数量果真可以利用代码度量进行预测？
- 设计模式果真有助于构建更好的应用程序？
- 人员个性会对结对编程产生何种影响？
- 地理位置的距离和公司职位的差距，究竟何者影响更大？

本书作者包括：

Jorge Aranda
Tom Ball
Victor R. Basili
Andrew Begel
Christian Bird
Barry Boehm
Marcelo Cataldo
Steven Clarke
Jason Cohen
Robert DeLine
Madeline Diep
Hakan Erdogmus
Michael Godfrey
Mark Guzdial
Jo E. Hannay
Ahmed E. Hassan
Israel Herraiz
Kim Sebastian Herzig
Cory Kapser
Barbara Kitchenham
Andrew Ko
Lucas Layman
Steve McConnell
Tim Menzies
Gail Murphy
Nachi Nagappan
Thomas J. Ostrand
Dewayne Perry
Marian Petre
Lutz Prechelt
Rahul Premraj
Forrest Shull
Beth Simon
Diomidis Spinellis
Neil Thomas
Walter Tichy
Burak Turhan
Elaine J. Weyuker
Michele A. Whitecraft
Laurie Williams
Wendy M. Williams
Andreas Zeller
Thomas Zimmermann

封面设计：Mark Paglietti 张健

图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

O'REILLY®
oreilly.com.cn



ISBN 978-7-115-27044-3
定价：89.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn